Autonomous Replication in Wide-Area Internetworks

Autonomous Replication in Wide-Area Internetworks

James Gwertzman

TR-17-95

April 1995

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

Autonomous Replication in Wide-Area Internetworks

A Thesis presented

by

James Gwertzman

to

Computer Science

in partial fulfillment of the honors requirements

for the degree of

Bachelor of Arts

Harvard College

Cambridge, Massachusetts

April 10, 1995

Abstract

The number of users connected to the Internet has been growing at an exponential rate, resulting in similar increases in network traffic and Internet server load. Advances in microprocessors and network technologies have kept up with growth so far, but we are reaching the limits of hardware solutions. In order for the Internet's growth to continue, we must efficiently distribute server load and reduce the network traffic generated by its various services.

Traditional wide-area caching schemes are *client initiated.* Decisions on where and when to cache information are made without the benefit of the server's global knowledge of the situation. We introduce a technique—*push-caching*—that is server initiated; it leaves caching decisions to the server. The server uses its knowledge of network topology, geography, and access patterns to minimize network traffic and server load.

The World Wide Web is an example of a large-scale distributed information system that will benefit from this geographical distribution, and we present an architecture that allows a Web server to autonomously replicate Web files. We use a trace-driven simulation of the Internet to evaluate several competing caching strategies. Our results show that while simple client caching reduces server load and network bandwidth demands by up to 30%, adding server-initiated caching reduces server load by an additional 20% and network bandwidth demands by an additional 10%. Furthermore, push-caching is more efficient than client-caching, using an order of magnitude less cache space for comparable bandwidth and load savings.

To determine the optimal cache consistency protocol we used a generic server simulator to evaluate several cache-consistency protocols, and found that weak consistency protocols are sufficient for the World Wide Web since they use the same bandwidth as an atomic protocol, impose less server load, and return stale data less than 1% of the time.

Contents

Chapter 1

Introduction

> Note: we are currently experiencing severe network problems on our 256 Kb/s school Internet connection. Please be understanding! I am still looking for a site willing to mirror the WebLouvre exhibit (30 Mb in all), preferably in the USA. —*WebLouvre*[31].

Traffic on the Internet[1] has been growing at an exponential rate for several years, but so far network technology has kept pace. There have been few growing pains, and bandwidth today is still plentiful despite annual predictions that the Internet is on the verge of collapse. We fear, however, that Internet growth is accelerating, and we believe that more efficient caching techniques are required to conserve Internet bandwidth and to allow Internet information servers to keep up with demand.

Table 1.1 summarizes the Internet's growth over the past few years, and also charts the growth of a relatively young Internet service, the World Wide Web [1]. Practically non-existent in 1991, the Web has grown in only 3 years to account for 23% of the total Internet traffic, 8% of that traffic in the last three months alone. The Web is popular, not only for the ease with which it allows information to be published, but also for the ease with which it allows information to be viewed.

Using Web client software, a click of the mouse can move megabytes of data across countless network hops. This ease of use has proven quite popular, and millions of new users are signing up to access the Web through online services.

[1] A glossary is provided for readers unfamiliar with certain technical terms used in this thesis.

Date	Total Traffic (Gb)	WWW Percentage (bytes)
12/91	1900	0.00%
12/92	4300	0.02%
12/93	8200	2.21%
12/94	16,000	15.98%
3/95	20,239	23.88%

Table 1.1: **Summary of Internet growth.** Notice that while traffic over the Internet only doubled last year, the percentage of that traffic for which the World Wide Web is responsible grew by a factor of 9.

The only problem with the Web is that it is inherently inefficient because it operates for the most part as a cache-less distributed system. When two clients retrieve a document from the same server, the document is transmitted twice, regardless of the two clients' proximity. This places more load on the server and uses more network capacity than is strictly necessary.

Some Web browsers have addressed this problem by adding local client caches. These caches store copies of data, allowing a client to re-access a previously viewed document without having to transfer it across the network again. Another approach to reducing server load and network bandwidth usage is the deployment of Web *proxies* [25, 29] that allow many clients to share one nearby cache. Since these clients all request their documents through the proxy cache, a given document only has to be transferred once across the Internet in order for many clients to view it.

The problem with both of these solutions is that they are myopic. A client cache does not reduce traffic to a neighboring computer, and a proxy cache does not reduce traffic to a neighboring proxy. Furthermore, these caches are typically limited in size: 5Mb for a client cache, and 100Mb-2Gb for a proxy cache are typical values. Disks might be inexpensive, but disk space will always be a limited resource, the growing use of multimedia on the Web means that files are growing larger, and local caches will only be able to cache the most popular items, resulting in many copies of the most popular items but no copies of only slightly less popular items.

A better solution would allow clients to share cache space cooperatively. The degree to which a file is replicated should be proportional to that file's global popularity, and objects should be cached where clients can reach them most efficiently. A system can achieve both goals by relegating caching

decisions to the server. A server has the ability to monitor the request stream and decide both when and where objects should be cached. In making these decisions, the server can take advantage of its understanding of the network topology and the file's access history to maximize bandwidth savings and reduce load. No existing caching technique satisfies all these constraints, however, so we found it necessary to create our own.

1.1 Push-Caching

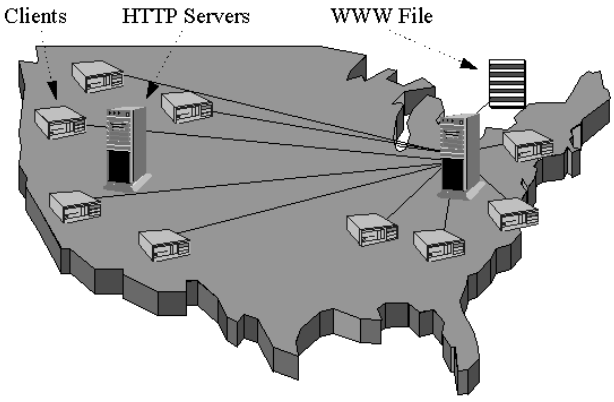
The goal of this work is to understand the access patterns observed in today's systems and evaluate a new, server-initiated caching policy that we have named *push-caching*. Servers record their access histories and use this information to decide where to place replicas of their data. We envision a network infrastructure with thousands of *push-cache servers* onto which files may be *pushed*. These push-cache servers may be created in an organized manner, but they may also be created in a grass-roots manner such as that which has driven Internet development to date. This infrastructure will of necessity be dynamic with push-cache servers added and removed on a continuous basis.

A centralized *registry* service tracks available push-cache servers, helping servers decide where to replicate their objects by providing a list of available push-cache servers on demand. This can be as simple as selecting n servers randomly from its list of all participating push-cache servers. The registry will be discussed more thoroughly in section 5.7.

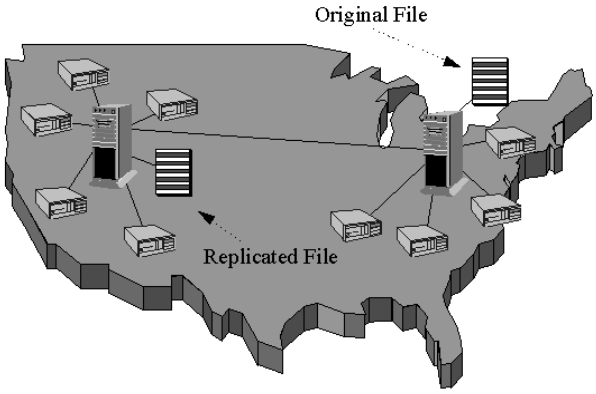
Figure 1.1 provides an illustration of push-caching in action, and demonstrates the two independent parts to a push-caching system. The server on the east coast experiences a great demand for its files, so it must decide when and where to replicate its files. Once the file is replicated west coast clients must efficiently locate the nearest copy of the files to access.

The optimal cache solution may be found with a perfect knowledge of network topology (i.e. which hosts are in close network proximity) and the access history of the documents in question. Topology information is not widely available on the Internet, however, so we also explore alternative ways to generate it using geographical information.

Although we limited this study to World Wide Web access, the algorithms and analyses are easily extended to any wide-area, distributed information system. Push-caching is especially applicable to applications such as video-on-demand that require large files to be efficiently and autonomously



(a) Before Push-Caching



(b) After Push-Caching

Figure 1.1: **Visualization of push-caching.** (a) Several clients accessing a World Wide Web file on the east coast. (b) The file has been replicated onto a west coast server so as to minimize network bandwidth, and now all west coast clients request the file from the west coast server.

distributed around a wide-area network such that bandwidth and latency are minimized. Server-initiated caching can also help solve the replication problems plaguing information services such as Archie [14], since we envision that scripts implementing these services may be replicated and distributed as easily as more traditional static objects.

1.2 Assumptions

We have made several assumptions in pursuing this work. We have assumed that the network on which this system will be built is unreliable, and that there is no guarantee that two arbitrary hosts will always be able to communicate. Given the unreliable nature of the Internet this is not a pessimistic assumption. We have also assumed a weak-consistency model for caching where data updates are not propagated to all replicas simultaneously; it is acceptable to occasionally provide stale data to clients as long as guarantees are made regarding how out-of-date data is allowed to become. We will prove the validity of this assumption in chapter 6.

Finally, we have assumed that ownership of a file is limited to one machine. Changes to a file can only be made by that file's owner, or *primary host*. This is a characteristic that distinguishes wide-area information systems from distributed file systems or other distributed systems with multiple write privileges. The primary host model simplifies the protocols because there is no need to grant or revoke write privileges, thereby helping to insure that push-caching will scale to millions of hosts.

1.3 Thesis Roadmap

Chapter 2 will discuss other research that has influenced our work, and place our investigations in context. Chapter 3 provides a motivation for our work, discussing several studies we performed to validate our initial assumptions. Chapter 4 discusses the trace-driven simulator we built to investigate push-caching in depth. Chapter 5 explores our results including a discussion of scalability. Finally, chapter 6 investigates various cache-consistency protocols, and chapter 7 concludes this work, summarizing our results and discussing the questions that still remain. Appendix A provides related works information that is important for distributed services but that does not directly affect push-caching.

1.4 A Note on Terminology

Throughout this document we will try to be consistent in our use of terminology. All of the various items available through the World Wide Web such as articles, pictures, files, etc. will be referred to as *objects*. The terms World Wide Web and Web will be used interchangeably, and they refer to the same thing. We distinguish between Web *clients*, Web *servers*, and *push-cache servers*. A server offers objects over the Web, and a client retrieves them. Servers push replicas of their objects onto push-cache servers, and push-cache servers may also push objects onto other push-cache servers. Finally, a *host* is any computer that is connected to the Internet: clients, servers, and push-cache servers included.

Throughout this document we will also make frequent references to the Internet; The Internet is a collection of many smaller networks, dubbed *subnets*. Subnets are connected to *regional service providers* that span specific geographic areas. Harvard, for example, is connected to NEARNet, the regional service provider for New England. These regional service providers are connected together by backbone networks: extremely high-speed networks that span the globe.

Chapter 2

Related Work

Little research has been performed directly on large-scale autonomous replication; only in the past few years have researchers begun assembling such systems [5]. There has, however, been a great deal of research on the various components of such a system, such as data caching and resource location. We anticipate that in the next few years we will witness a convergence among distributed systems research toward large-scale autonomous replication as large-scale systems such as the World Wide Web become more ubiquitous. Traditional client caching, for example, has slowly been giving way to more aggressive types of client caching, and recent cache consistency research has focused on increasingly sophisticated ways to update caches.

This chapter discusses current research into the various fields that have inspired large-scale autonomous replication. Section 2.1 examines distributed file systems, and section 2.2 considers recent applications of older distributed file system research. Section 2.3 covers research into locating nearby Internet resources of interest, and finally, section 2.4 discusses recent Internet surveys. We delay analysis of cache consistency protocols until chapter 6, where we examine consistency research in greater detail, and we include a description of group-communication research in Appendix A. This research does not directly apply to the work performed for this thesis, but is necessary for efficiently maintaining a large, shared database such as that used by the proposed registry of push-cache servers.

2.1 Distributed File Systems

Much of the current research in autonomous replication has been heavily influenced by the caching subsystems of distributed file systems. Efficient operation of a large-scale distributed file system depends on caches to reduce the load on file servers; systems like Sun's NFS [27, 26] that do not rely on long-term caches cannot support more than a few hundred clients, whereas systems like xFS [35] that distribute server load among clients are designed to scale to many thousands of clients.

The challenge of offering distributed access to a wide-area information system is almost identical to that faced by a large-scale distributed file system; some researchers have proposed making this relation explicit [33] by actually serving files from the Web over a wide-area distributed system like the Andrew File System [28]. This idea is still contentious, because the semantics of the World Wide Web may not match those of a distributed file system. File systems, for example, need to rapidly create and destroy very small files that will never be shared by multiple users. The Web, on the other hand, needs to provide rapid access to many files from across the entire Internet. Very few of these files are local. We can nevertheless learn much from the distributed file system community about building large-scale systems since many of the scalability concerns are the same. In the next section we discuss the large-scale distributed file system designed by Blaze as part of his doctoral dissertation.

2.1.1 Large-Scale Hierarchical File Systems

The goal of Blaze's thesis [4] was to design a distributed file system that could scale to a very large size, operating across the Internet. He achieved this goal by building a hierarchical system where clients can not only cache items indefinitely but can also serve them to other clients.

Blaze introduces four types of scale that a successful system must address: population, traffic, administrative, and geographic. Each type has its own set of issues. Scaling with population requires a system to cope with growth in the total number of potential clients and implies that a server should not need to store any information about its clients. Scaling with traffic requires the ability to handle the workload generated by all clients. Scaling administratively implies an ability to span autonomous entities. Finally, coping with geographic scale requires the ability to span large distances, with the inherent latency implied therein.

The Web in its current form addresses these by eliminating many of the frills associated with distributed file systems such as caching and write-sharing. Our challenge is adding an optimized caching system to the Web without reducing its ability to scale along these lines.

Before designing his system, Blaze gathered traces from an NFS server to determine optimal cache strategies for his distributed file system. The most important trend that emerged is that files tend to display strong *inertia*, meaning that the same files tend to be opened again in the same mode by the same user. Blaze found, for example, that keeping a file in a file system client cache for two hours resulted in an average hit rate of over 66%.

Blaze also found that most files are overwritten while still very young. If a file is not changed soon after creation, the probability of its ever being changed drops quickly. Files therefore move rapidly toward a state of being read-only, and Blaze found that shared read-only files are even less likely to be written to than private read-only files. Finally, files opened for reading by other machines are likely to be opened for reading by still others. These results imply that cache sharing will work well for a distributed file system because it is unnecessary to update shared-read files often. We will see that a similar analysis drives the Alex FTP cache as well, and in chapter 6 we will see that these results also apply to the World Wide Web since Web files that are globally popular changes less often than those that are primarily used locally.

Blaze analyzed a variety of different caching schemes. Flat file systems, where all clients connect to the same server, inevitably suffer from a bottleneck effect. Eventually the single server is unable to cope with the connected clients. Even with optimal client caching strategies the server must still deal with 8-12% of the original traffic, and as the system grows this will eventually overwhelm any server. Such systems include not only file systems such as NFS, but also the existing World Wide Web.

The solution is to distribute load among several servers. This is traditionally done through a fixed hierarchy, where a client does not request files directly from the primary host, but rather from one of several intermediate servers. If the intermediate server does not have the file it forwards the request for the file on to the primary server.

There are problems with this solution, however. Replicating all primary server files on the intermediate servers is inefficient, but caching items at intermediate servers as requests pass through them creates delays. Furthermore, having an intermediate cache process all requests yields surprisingly little gain. Most objects not in the client cache were not in the intermediate

cache either. The answer is to look in the caches maintained by other clients: 60-80% of client cache misses were of files already in another client cache.

Blaze's architecture is similar to our push-cache architecture. In his system, servers record the identity of clients who request a given file. When a file receives too many requests the server stops serving the file directly and instead returns the list of clients who have accessed (and therefore cached) the file. Clients use this list to locate nearby replicas of a given file, storing the list of other clients in a name cache. We use a similar method, described in section 4.3.4, to locate replicated files on the Web.

We are unable to apply Blaze's work directly to the Web because it makes assumptions about clients that we cannot: that clients can function as servers, and that clients have sufficient cache space to cache files on a long-term basis. The first assumption is false because the majority of clients on the Web are personal computers that are not available 24 hours a day. The second is false because client caches are typically small, and because the Web encourages a browsing behavior that results in a rapid cache turnover of files. We improve on Blaze's work by not only removing these assumptions, but also by determining the optimal location for caching file replicas rather than relying on the arbitrary list of clients who have cached the file.

2.1.2 Alex

Another system from which we draw inspiration bridges the gap between data replication and distributed file systems and is called Alex [8]. Alex allows a remote FTP server to be mapped into the local file system so that its files can be accessed using traditional file operations. This is accomplished through an Alex server that communicates with remote FTP sites through FTP, caches files locally, and communicates with the local file system through NFS. When the user wants to access a remote file, Alex downloads the file using FTP and then caches the file locally.

Alex's primary contribution is its cache-consistency mechanism. As we have already seen from Blaze's thesis, the older a file is the less likely it is to be changed. Therefore, the older a file is, the less often Alex has to poll to insure that its local copy is up-to-date. To avoid excessive polling, Alex only guarantees that a file is never more than 10% out of date with its reported age. As an example, if a file is 1 month old, then Alex will serve the file for up to three days ($10\% \times 30$ days = 3 days) before checking to see if it is still valid.

This is efficient, not only because FTP servers do not have to propagate

file updates, but because they do not have to be modified to perform more sophisticated invalidation techniques. In chapter 6 we find that this cache-consistency scheme is applicable to the World Wide Web as well as to FTP. In the next section we examine Web proxies that extend the concepts of Alex to the World Wide Web by caching Web data just as Alex cached FTP data.

2.1.3 World Wide Web proxies

A World Wide Web proxy [25] caches Web data for a campus-sized network. They are also used by corporations who are protected by an Internet firewall and whose computers therefore cannot access the Web directly. The proxy accepts requests for Web documents from clients, retrieves and caches the documents, and then makes them available to its clients. When a client wants a document that is already in the cache the proxy can serve it without having to re-retrieve the same document.

The most popular Web browsers support proxies [29], and the HTTP protocol has provisions to help maintain cache consistency. If the proxy wishes to determine if its cached page is up-to-date, a lightweight protocol exists known as the *Conditional GET*. A browser may make a `get` request to the server that includes a timestamp, the *If-Modified-Since* field. If the page has been modified since that time, the server will re-send the page. Otherwise the server will respond with a *Not Modified* message. Some proxies use a similar cache-consistency protocol to that used in Alex.

Netscape, a company that sells a commercial Web proxy, claims that a 2 or 3 gigabyte Web proxy can support thousands of internal users and can provide a cache hit ratio as high as 65%. Results such as these indicate that Web proxies can help alleviate some Web scalability concerns, and we will compare push-caching to proxy-caching in section 5.6. As we saw above with Blaze, however, even optimal proxy-caching will not help distribute server load sufficiently since proxies must still satisfy cache misses from the primary host.

2.2 Internet Caching Schemes

We are not the only researchers turning our attention toward saving Internet bandwidth. Another group that has been particularly vocal on these issues consists of Bowman, Danzig, Manber, and Schwartz [5, 12, 10, 11].

These researchers believe that the ideal caching architecture for the Internet is hierarchically structured, and they have already built a system called Harvest (see section 2.2.2) to test their ideas. We believe that their system will suffer from the same limitations that caused Blaze to turn away from hierarchically structured caches, but they do not yet have data to confirm or deny our belief.

We agree with the Harvest group on other issues, however. They have stated that servers should be instrumented to help determine where to place additional replicas; we take that belief to its logical conclusion by designing a system where servers are not only instrumented, but can replicate their most popular objects without human intervention. The fact that our system is capable of this autonomous replication illustrates the primary advantage of a server-initiated caching system over a client-driven one such as their Harvest system.

The rest of this section explores the Harvest group's research in greater detail.

2.2.1 Caching FTP Objects

The primary motivation for the Harvest group's hierarchical cache is a study by Danzig, Hall, and Schwartz that shows that FTP traffic across the Internet backbone could be reduced by 42% if file caches were installed in a hierarchical manner at strategic locations on the Internet [10]. Specifically, they propose placing FTP caches at all juncture points between networks, such as between a backbone network and a regional network. Cache resolution would take place in a hierarchical fashion, with each cache satisfying cache misses from its parent, and so on, until the file is eventually retrieved from its home FTP server.

To compute the bandwidth savings of this caching, they used the metric of *byte-hops* where the cost of a FTP transfer is the size of the file in bytes times the number of network hops the file had to make. This is a useful metric for computing network cost, and we will use this metric in our own simulations.

Their results show a 42% reduction of FTP traffic across the Internet backbone by adding these hierarchical caches, but we question the need for a hierarchical structure on the Web given the 65% bandwidth reduction of simple proxy-caching. For one thing, regional network level caches will need to be very large. For another thing, installing these caches in between regional networks and backbone networks will not be trivial, and it is unclear

who should manage these caches. Recall that Blaze stated that hierarchical caches are often not effective; the performance of Web proxies in the absence of any sort of hierarchical caching seems to justify this hesitation.

2.2.2 Harvest

The Harvest system [5] was designed to solve the problem of resource location, but also includes a hierarchical caching subsystem. This caching subsystem functions like a Web proxy in that clients request Web objects through the local Harvest cache. If the object is not in the cache, then the local cache in turn queries each of its neighbors, its parent, and the object's primary host looking for the object. Whichever host returns a "hit" fastest is the host from which the cache requests the object.

The most interesting feature of this system is that the client requests data from the host that responds most quickly. This is significant because recent research from Bestavros' group at Boston University shows that latency on the Internet between two hosts varies significantly, even over short time periods. If the goal is to reduce the latency seen by the client then this approach might be the most successful. Problems with the Harvest system have already been discussed; specifically, the requirement that Harvest caches be deployed hierarchically.

2.3 Resource location

The final component to large-scale autonomous replication is efficiently locating the nearest replica of a given file. It is easy to make a copy of a piece of data; deciding which copy to use is difficult. Resource location, for example, was the primary difference between Blaze's distributed file system and a traditional distributed file system. Under his system, it was not necessary to satisfy a cache miss with the primary host. A host could locate a copy of a file in another host's cache.

Push-caching is similar to Blaze's system in that cache misses can be satisfied out of other caches; it is different in that the locations of these caches are computed so as to minimize network traffic, and cache misses must be satisfied out of the closest cache. Our resource location scheme will therefore need to be able to locate the closest copy of the file.

Guyton and Schwartz are interested in discovering a nearby resource without any sort of centralized database whatsoever [19]. This differs from

earlier approaches such as that used in Grapevine for example [3] which required centralized shared databases.

2.3.1 Locating Nearby Copies of Replicated Internet Servers

Guyton and Schwartz try to determine how to choose among a collection of replicated servers such that the selection takes into account network topology [19]. They evaluated a variety of approaches using a network simulator, uncovering a number of tradeoffs between ease of deployment, effectiveness, network cost, and portability. They finally conclude that there is no obvious "best approach," but only a variety of compromises.

At the heart of this research is the fact that in the current Internet there is no magic black box to determine Internet topology. If this information were known, then optimal resource location would be not only possible but trivial, because this global Internet topology map could be consulted to determine exact host distances. The purpose of Guyton's and Schwartz's research is to determine the cost and effectiveness of approximating this information through various means; in section 3.3 we extend this research further by determining how well geographical information approximates Internet topology.

Guyton and Schwartz examine the variety of choices that distinguish between various resource discovery approaches, and conclude that each of these choices lies on a spectrum with ease of deployment/high network cost on one end, and difficult deployment/low network cost on the other. None are optimal, and only the least accurate scale in a manner appropriate to the World Wide Web. Route probing, for example, one of the most accurate methods, requires a measurement server to calculate the shortest path in a dense graph, a non-trivial calculation. If the effort is multiplied by the thousands of clients that would need to use such a service this option becomes infeasible. The fact that an efficient means for detecting Internet topology does not exist forces us to turn toward more radical solutions, such as using geography to predict topology.

2.4 Internet Analysis

No distributed wide-area system can be built without an analysis of the underlying Internet strata. A designer must understand the performance issues of the Internet in order to build as efficient a final product as possible.

There have been a number of Internet studies; we describe here the most recent reports.

2.4.1 Web Traffic Characterization

The Applied Network Research Group at the San Diego Supercomputer Center has been researching the Internet for years. They have recently turned their attention to the Web and they analyzed access logs from the NCSA server, a globally popular Web server, to show that geographic caching helps reduce network traffic, latency, and server load [7].

The ANRG's research anticipates our own research since they provide motivation without implementation specifics. They found that server overloading is a growing problem for Web servers, but are afraid that solving the load problem will lead to more file requests which in turn will aggravate the network bandwidth consumption. Any solution to load balancing must distribute load so as to reduce the network bandwidth requirements.

The ANRG suggest that to help distribute loads, client preference for a cache site might be a function not only of location, but also of current network or server load. To determine how much network bandwidth could be saved by caching files geographically they performed a simulation driven by Web access logs, mapping Internet addresses to states or countries in order to determine geographic location.

The ANRG's proposed model divides the Internet into specific geographic regions and places Web caches in each region. They evaluated the bandwidth savings due to such a system by assuming that all clients in a given region use that region's cache to satisfy their requests. The efficiency of geographic caching is computed using an efficiency factor of bandwidth savings in bytes divided by total cache size. Their simulations achieved at most an efficiency factor of 7; we show in section 5.6 that push-caching can achieve efficiency values of 700 or more, since push-cache servers know how popular the items are that they are caching, and therefore can set the amount of replication accordingly.

The ANRG also suggest that one way to solve the resource location problem is to modify a system called the Domain Name System (DNS) that is currently used on the Internet to translate between human-readable machine names and computer-readable machine names. They suggest adding to DNS the ability to look up nearby replicas of objects or services, defining "nearby" to include metrics like physical distance and number of network hops. Finally, they mention that no caching solution is complete without

taking into account file security and different levels of cache time-out for different types of data. We are particularly interested in the possibility of a modified DNS because it would allow us to avoid contacting the primary host in order to locate a nearby replica.

2.4.2 Web Availability and Latency

Viles and French studied the availability and latency of Web servers on the Internet [34]. Ideally all servers should be available 100% of the time, and should have latencies of 100ms or less. They found that in reality the Web does not live up to these ideals; most servers are on average only available 95% of the time, and the average latency is much higher, on the order of 500ms.

Viles and French suggest one possible way to improve server latency: implement a way for the client to request several documents at one time. This amortizes the cost of the TCP connection setup and takedown times over several documents instead of just one, since TCP costs turn out to be an important factor for short documents.[1] They do not address the availability issue further.

Both of Viles' and French's findings have bearing on our work, because autonomous replication should be able to solve both the availability issue and the latency issue. By replicating objects availability is improved, because a single server failure no longer eliminates all access to that server's objects. Latency is improved as well if network topology and access history are both considered when deciding where to cache objects. When nearby object replicas are available clients will observe decreased latencies in accessing those objects relative to accessing more distant replicas.

[1] The TCP protocol is used to insure reliable packet delivery across the unreliable Internet.

Chapter 3

The Case for Push-Caching

Client-based caching is popular on the Web because it is easy to implement, and because it provides significant bandwidth and latency savings. We are investigating push-caching because there is, unfortunately, an inherent limit to the amount of bandwidth and server load reduction possible with client caching.

Even if every Web site was using a Web proxy, it would still be possible for a site to become swamped if a large number of proxies try to access a specific object at the same time. This is not a rare condition; a current Web service known as the `Cool Site of the Day` is very popular [13]. Each day it lists a "cool" Internet site that subsequently receives so much traffic that it often becomes swamped. Push-caching is initiated by the server; it alleviates this problem by autonomously replicating the "cool" site's data when load at that site dramatically increases. Push-caching complements client-based caching by helping to disperse server load; together the two can provide more efficient network transport of data than either can provide independently.

Optimal push-caching is only feasible on the current Internet if it is possible to derive reasonably accurate network topology information from the chaotic, unordered Internet. As explained in section 2.3.1, Guyton and Schwartz showed that it is impossible to directly derive this information efficiently from the Internet itself. One goal of this work is therefore to demonstrate that geographical distance, which is easy to derive, predicts topological distance. We also show that a dynamic replication protocol like push-caching is needed because simpler caching solutions, such as a static mirroring of popular Web sites, do not suffice.

3.1 Web-Trace Analysis

We collected traces of several different Web servers to answer the following questions:

- Are access patterns sufficiently skewed that caching a small number of objects will satisfy a majority of Web requests?
- Are access patterns inconsistent enough that simply mirroring popular Web sites will not provide good server and network utilization?
- How accurately can a server derive Internet topology, thereby approximating the optimal case? Does this topology also relate to Network latency?

We selected four Web sites for trace collection: the globally popular `www.ncsa.uiuc.edu` (home of the Mosaic browser), our locally popular `fas.harvard.edu` (a campus-wide information server), `hcs.harvard.edu` (a computer club's server), and `das-www.harvard.edu` (the computer science department's server).

We modified the three servers on our campus to record when the object being transferred was last modified. This information is not usually logged, but is essential for performing an accurate simulation of consistency mechanisms. We use this information in chapter 6 to realistically simulate cache consistency issues.

Of the four traces we used to drive our simulator, the NCSA trace is most representative of the globally popular servers that our algorithms are designed to help. The other three traces are more useful for exploring the effect of server-initiated caching on the less popular but more numerous small-scale servers.

Before we began analyzing these traces we had expected that some pages would be much more popular than others, if only because most Web sites have a "home page," or table of contents, that lists the contents of that server. There are usually several other files associated with this home page, and almost every visitor to a Web site sees these files. These pages, at least, will be exceptionally popular relative to the rest of the Web site. We also expected that access patterns would not be consistent across servers; some popular servers such as the Boston Restaurant Guide [9] or New England Alpine Ski Report [32] have specific geographic interest, while others such as the White House home page [20] have a uniform appeal. Finally, we

Server	Traffic/day (Mb)	Requests/day	% Static	Files	Time-Span
NCSA	6,492	605,454	66%	5610	1 day
FAS	9.3	2,760	73%	292	1 month
HCS	8.1	1,750	72%	575	1 month
DAS	15	1,390	73%	1405	1 month

Table 3.1: **Summary of the Web server access traces.** The Requests/day column indicates the average number of requests that appeared in the trace for that server per day, while the % Static column indicates that percentage of the requests that are valid and for static objects (as opposed to dynamic pages for example). The Files column indicates the total number of documents that are cacheable on the server. Note that the NCSA trace is for one day, while the other three are for one month.

expected geography to predict topology to some extent—there should be more network hops between a site in California and a site at Harvard than between a site at M.I.T. and the same site at Harvard.

Our trace analysis revealed the following facts:

- On average, 75% of Web requests are for 4% of a server's files.
- The more globally popular a server is, the smaller the set of popular documents.
- Requests to popular servers are not geographically uniform; different servers have different access patterns.
- The location of past requests predicts the location of future requests.
- The number of network hops and network latency are strongly correlated to each other as well as to the geographical distance between two hosts.

In the rest of this section, we will explore each of these facts in greater detail.

3.1.1 Popular Files are Very Popular

First, we examined the distribution of Web accesses per object. This distribution is shown in Figure 3.1. As we had expected, access patterns are

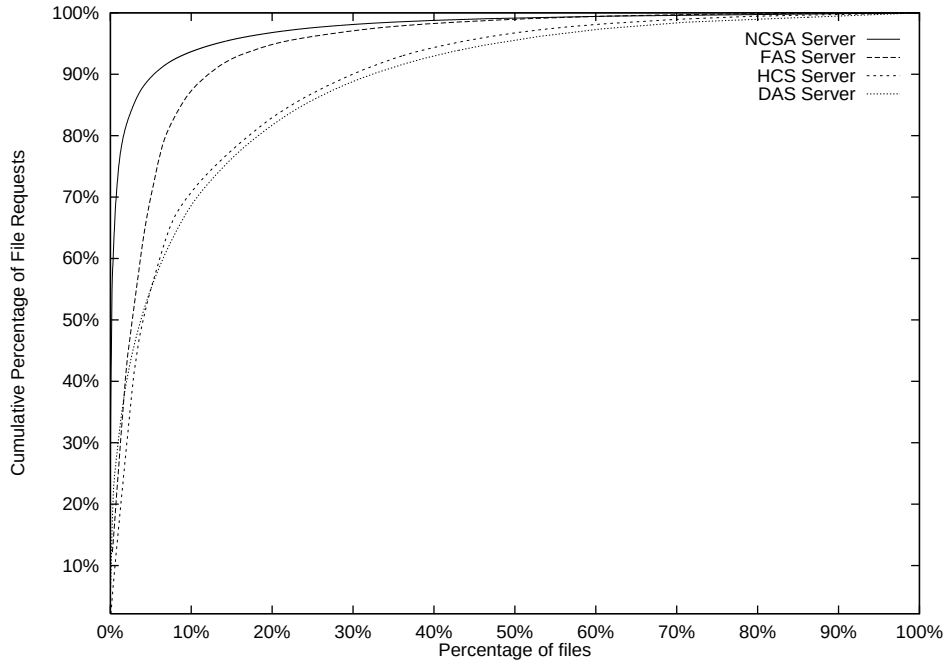


Figure 3.1: **Popularity analysis of requested files from four different Web sites.** The skewed distribution of requests indicates that caching a minority of the files can satisfy a majority of the requests. Files are ordered by their popularity.

highly skewed. The graph indicates that a small percentage of the files available on a given server are responsible for a disproportionate share of the requests from that server. For example, the top 5% of the files on the NCSA server were responsible for 90% of the total requests from that server.

Bestavros confirms these results [2], adding that the more globally popular a server, the smaller the fraction of pages that account for most of its accesses. Our results agree with this observation: the two most popular servers, NCSA and FAS, are also the two with the smallest percentages of files responsible for the most requests. These results are encouraging because they suggest that caching a small subset of a server's files will reduce the server's load significantly.

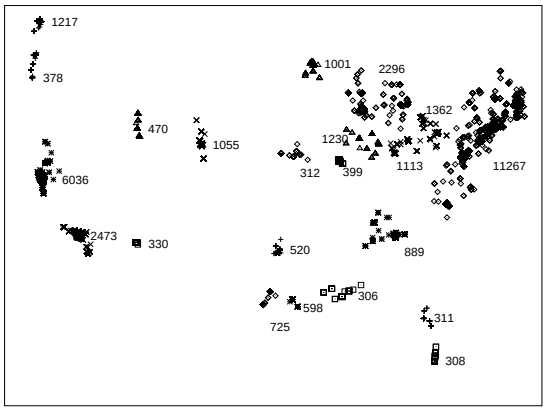
3.2 Static Mirroring is Unlikely to Work

If server popularity were a continuous phenomenon and requests were always clustered in exactly the same way, then mirroring popular Web sites in a static manner would be optimal, and clever caching algorithms would be unnecessary. Unfortunately, this does not seem to be the case. Using a *friends-of-friends* algorithm, popular with astrophysics researchers for detecting galactic clusters, we performed a geographical cluster analysis on our traces to determine groups of hosts with common access patterns. The algorithm is simple: if a host is within a certain distance d of any member of an existing cluster, then the host joins that cluster. If a host joins multiple clusters, then those clusters are merged together. If the host can not join any cluster, then the host becomes a member of a new cluster.

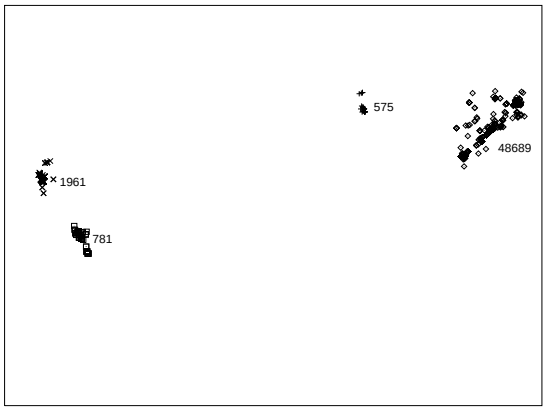
For the algorithm to work properly, d must be set such that the clusters are neither too big nor too small. Through trial and error we found that a range of 50 miles works well for the Internet. Figure 3.2 displays a cluster analysis of the NCSA and the local FAS traces.

Two results are apparent. There are prominent clusters in both California and New England on the two graphs; however, the distribution of requests differs significantly. Secondly, the more popular server has many smaller clusters in addition to those in New England and California. These results indicate that no single server scheme would be able to cache pages for multiple, unrelated servers efficiently. Mirroring our campus information server on the opposite coast might be helpful, but such a simplistic answer would not help the NCSA server, which is popular in 22 separate clusters. On the contrary, a scheme that can autonomously replicate the NCSA server's most popular files in 22 different geographic regions while replicating our campus server in only 4 will be more efficient. Furthermore, dynamic replication schemes also help distribute server load more efficiently.

The graph makes it easy to see the intuitive advantage of distance-sensitive caching: the subnetworks represented by each point lie tantalizingly close to each other. Proxy-caching requires a separate cache on each subnetwork, but distance-sensitive caching enables the entire cluster to autonomously share one local cache.



(a) `www.ncsa.uiuc.edu`



(b) `fas-www.harvard.edu`

Figure 3.2: **Cluster analysis of the globally popular `www.ncsa.uiuc.edu` and our locally popular `fas-www.harvard.edu`**. Each point represents an entire subnetwork plotted geographically; the numbers indicate the total number of requests received from each cluster. Clusters with fewer than 300 requests have been omitted. Notice that the ratio of requests between the East coast and the West coast for the FAS server is roughly 18 (a), whereas the ratio for the NCSA server is roughly 1.6 (b).

3.3 Deriving Network Topology

For a server to make the optimal decision about where to cache data it must have an accurate representation of network topology. As we saw in section 2.3.1, there is currently no way to determine *á priori* the Internet's topology. We hypothesized that geographical information could be used to hint at which servers were topologically close.

We surveyed the Internet using the `traceroute` [21] program to measure Internet topology, and we used a file maintained by Merit [30, 7] listing the address of each subnet administrator for the 42,000 subnets on the Internet today to measure geographic data. The critical datum in the Merit file is the zip code listed in the address; in conjunction with a geography server [24], this provides enough information to establish the latitude and longitude of each network administrator. As long as the zip-code of the subnet administrator matches the zip-code for the subnet as a whole we can accurately place the subnet geographically. It is a simple calculation using this information to compute the distance between two arbitrary hosts on the Internet, accurate to within a zip-code and the size of the subnet. This approach is not effective for subnets that span multiple zip-codes, such as backbone networks or regional networks, but it is effective for the local networks that account for a large fraction of the client requests.

To test the correlation between these two types of information we selected several hundred hosts in the United States and surveyed each one's distance from Harvard. We calculated the latency between our host and it, as well as the number of network hops between them using `traceroute`. We also calculated the distance between them in miles as described above. Since individual workstations are frequently not accessible our survey settles for any computer it can reach on the same subnet as the desired host. If it is not possible to access any host on the desired subnet then a failure is recorded for that host. We ran this program from several other locations around the Internet, including the west coast and Colorado.

We did not expect extremely high correlations because Internet connectivity varies widely. While some hosts are connected by a high-speed network connection, other hosts are connected by slower, less well-connected networks. Different backbone connections are another source of error; because these backbones only connect to one another at a few sites, a file exchanged between two hosts on different backbones, no matter how close to each other geographically, may have to travel quite far on the Internet. As an example, the hosts `maddog.harvard.edu` and `carrara.bos.marble.com`,

Backbone	Sample Size	Hops: Miles	Latency: Miles	Backbone Hops: Miles	Hops: Latency
All	314	0.3927	0.2400	0.4450	0.4243
204.70	185	0.7394	0.5781	0.7966	0.7340
140.222	64	0.6365	0.4270	0.8535	0.7317
144.228	45	0.4497	0.1146	0.4867	0.4727
137.39	15	0.0568	0.2194	0.1640	0.4468
134.55	5	0.6217	0.9790	0.5468	0.6374

Table 3.2: **Backbone-based correlations for geographical distance versus network hops, latency, and backbone hops, as well as network hops versus network latency.** We have divided our samples into groups based on the backbone to which they are connected. Measurements were taken from a host on the 204.70 backbone (the NSFNET); notice how correlations are strongest overall for other hosts on 204.70. Correlations were computed using a power-law regression fit.

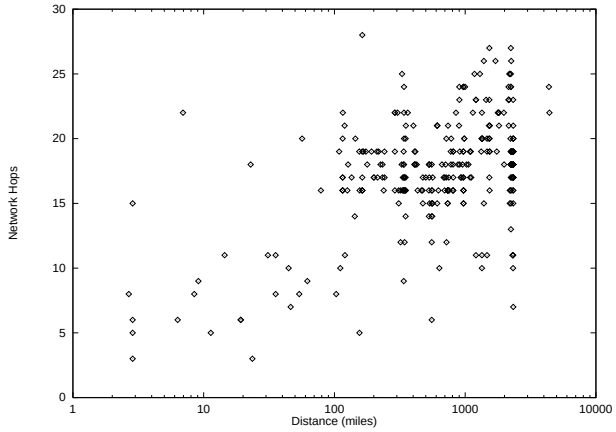
are both located near Boston, but since one is on the MCI backbone and the other is on the Sprintlink backbone packets between them must pass through Washington, DC where the two backbones connect.

Figures 3.3 and 3.4 display the data from our east coast observations for distance versus network hops, distance versus latency, distance versus backbone hops, and network hops versus latency. The Colorado and west coast observing runs yielded similar results.

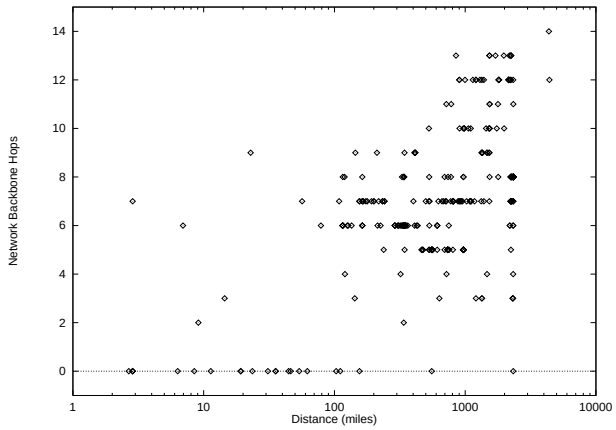
In looking for signs that geographical distance predicts network distance (network hops, backbone hops, and latencies), we were encouraged by the apparent correlation shown in the graphs. We also noticed the trend that nearby hosts show the greatest correlation between geographic distance and network distance. Once the distance exceeds 500 miles, the importance of geographic distance decreases.

We hypothesized that if we limited our analysis to hosts on the same backbone network, we would find stronger correlation between geographical distance and network distance. Table 3.2 presents the results of this study. To examine this hypothesis we divided the hosts into several groups, one group for each backbone, and then computed the correlations for each backbone separately. The correlations were computed using a power-law regression fit.

These observations affirm our hypothesis that the correlations between

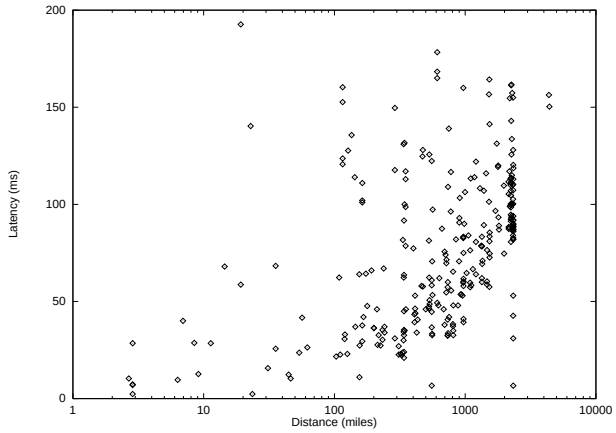


(a) Network Hops vs. Distance

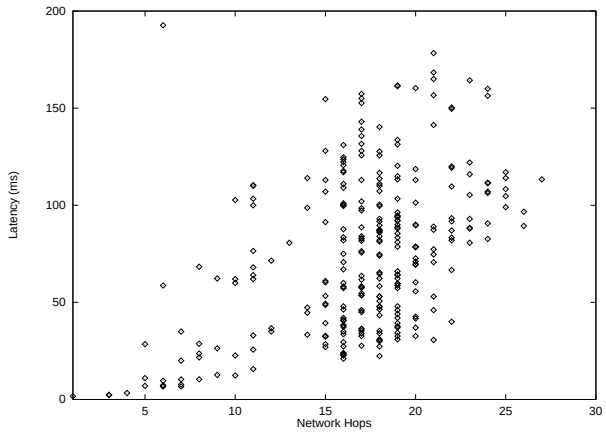


(b) Network Backbone Hops vs. Distance

Figure 3.3: **Results of Network Survey: Network Hops and Network Backbone Hops.** Note that geographical distance establishes a lower bound for network hops. Note also the number of hosts in the sub-100 mile range that are 0 backbone hops away.



(a) Network Latency vs. Distance



(b) Network Backbone Hops vs. Latency

Figure 3.4: **Results of Network Survey: Network Latency.** Note that the latency graph was cropped at 200 ms for clarity; there were 17 hosts with latencies ranging from 200ms to 1s that were removed.

geographic distance and Internet distance are higher overall when looking at hosts on the same backbone than when looking at all hosts. This result suggests that it will be advantageous to steer clients toward host caches that are both geographically close and on the same backbone network.

We included a comparison of network hops to latency because calculating expected latency on a network is hard. It requires a knowledge of network bandwidth and expected loading. If the number of network hops between two computers is related to the latency, then by optimizing to reduce network hops we are also optimizing to reduce latency. Figure 3.4 indicates that there is a moderate correlation between hops and latency: fewer than average hops implies low latency, and more than average hops implies a high latency. This is helpful, because it implies that steering a host from a distant cache to a close cache will decrease latency as well as network traffic. As we saw in section 2.4.2 this should be one of the primary goals of replication schemes.

We investigated latency further by following up on a suspicion voiced by Bestavros in a private conversation. He suspected that latency was primarily caused by crossing between backbones, not necessarily by the number of individual backbone hops. This hypothesis would make sense if connection points between backbones proved to be bottlenecks and sources of congestion. We therefore modified our survey to also include the number of backbones traversed. The results of this new survey are in Figure 3.5. There is a clear correlation between the maximum latency observed and the number of backbones crossed, although we can not draw any further conclusions from the data. We hope to follow up on this finding in future work; if it turns out that latency is strongly related to the number of bandwidth hops then simply mirroring web sites on multiple backbones should reduce latency considerably.

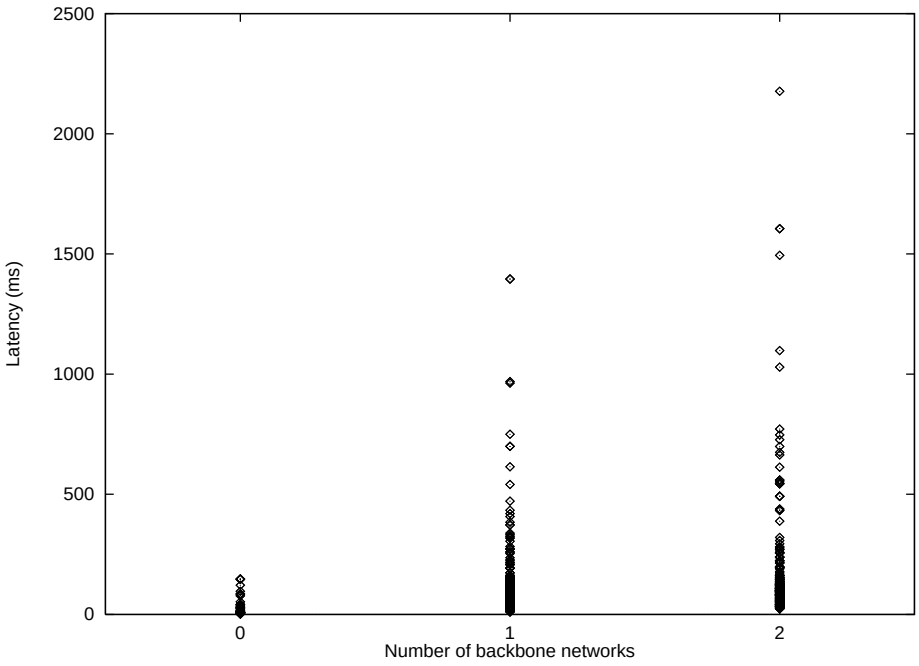


Figure 3.5: **Number of backbones versus latency.** There is a clear correlation between the maximum latency observed and the number of backbones crossed. This would support Bestavros' suspicion that crossing backbones accounts for the majority of the Internet's latency. Notice that no latencies greater than 100 ms were observed without crossing at least one backbone boundary.

Chapter 4

Experimentation

The true test of any theory is a real world implementation. Only by actually constructing and distributing a working Web push-cache system can we accurately assess its performance. Unfortunately, such an approach leaves no room for evaluating the various parameters that exist for such a system. Once distributed, any system becomes hard to change.

We decided therefore to first build an Internet simulator that would allow us to evaluate push-caching and to compare it with other popular caching methods. No simulator is perfect; simulated results are only useful insofar as they provide relative performance between competing schemes. Furthermore, in designing such a simulator it is very important to avoid building any bias into the simulator that might influence its performance. We were careful to be objective in building the simulator, gathering trace data, and setting the parameters that determine its performance.

4.1 Simulator Design Goals

Before we could design our simulator, we needed to decide what was to be simulated. What questions were we trying to answer, and what data would the simulator need to provide to help us answer those questions? The simulator would be used to compare the relative performance of various caching schemes; therefore, it needed to provide data that could be used to judge relative performance. This includes: network bandwidth consumption, network latency, cache hit-rates, and resources consumed such as server load and disk space.

We decided to build a general simulator that would simulate the Internet

itself. Our caching scheme could be implemented on top of this Internet simulator, and if we wanted to experiment with different approaches we could do so without having to build a separate simulator. This would eliminate any potential bias due to simulator differences.

4.2 Simulator Implementation

Our simulator is divided into three parts; the simulator engine that drives the simulation, the host modules that implement the caching schemes, and the network module that simulates the Internet.

The engine reads in the trace data, and uses it to create events. Each event represents a host requesting a World Web document from a server; the actual event consists of the server and host involved, the name of the document, the size of the document, and the date and time of the transaction.

The engine also provides support for periodic jobs that must be run on a frequent basis. When a job is scheduled to run it is removed from the queue and the associated code is run. The code may, as part of its duties, re-insert the job on the queue. This allows us, for example, to reset a host's load every hour, and to print out simulator statistics every six.

There are two different types of host modules: servers and clients. Clients are handed an event by the engine and use this information to request a Web document from a server. This request travels through our simulated Internet where the appropriate bandwidth is logged. Servers have all the functionality of clients, but can also serve files. Servers keep track of which files they are caching; one server is designated as the primary host and maintains the most up-to-date versions of files. Both types of hosts record a variety of load parameters such as number of requests, number of bytes stored, number of bytes sent, client-cache hits, and stale client-cache hits.

Depending on the exact caching scheme being simulated, the server can return either the requested document, or the name of another server in the form of a redirect message containing a list of documents currently replicated at that server. Clients can cache not only redirects, but also files themselves in order to simulate client-caching. Servers can also push documents onto other servers; a centralized manager keeps track of servers willing to accept documents.

The Internet module was the most difficult portion of the simulation to build. We wanted it to log not only the number of packets passing across

it, but the exact bandwidth used in terms of bytes $\times$ hops, where 1000 bytes sent across 10 network hops would use twice the bandwidth of 1000 bytes sent across 5 network hops. In order to log bandwidth accurately the simulator needed an accurate Internet topology, but it would clearly be impossible to calculate and store the entire Internet topology. Even if we limited the amount of information we needed to the 60,000 or so hosts mentioned in the NCSA access log we would still be faced with an impossible amount of information.

We also did not want to synthesize topology information because push-caching is very sensitive to the interaction between geographic information and Internet topology. If we used the same assumptions to generate the topology that push-caching used to approximate it we would see artificially high correlations between predicted performance and actual performance, a clear case of simulator bias.

We were fortunate therefore to be given access to the Internet topology gathered for a Network Time Protocol server survey by Guyton and Schwartz [18], discussed in section 2.3.1. Using this topology we were able to accurately measure the number of hops between two arbitrary hosts, and thereby compare the efficiency of using miles to predict topology versus using the actual topology itself. We were unable, however, to accurately measure the number of backbone hops between two hosts or to calculate the latency between two hosts using this data. These are factors that we can only crudely approximate using the results of section 3.3.

4.2.1 Simulated Internet Topology

Of the 6700 hosts in Guyton and Schwartz's database, we were left with 1700 hosts after removing network routers and hosts on networks that span multiple zip-codes (which we could not therefore locate geographically). Figure 4.1 displays the hosts graphically, plotting each host's longitude and latitude. It is easy to see the distinctive shape of the United States, as well as the presence of hosts in Alaska and Hawaii.

Since our results rely so heavily on this data we took several steps to assure its authenticity. A host from Harvard was conveniently included in the study; we calculated the number of network hops from this host on the simulated Internet to all the other hosts on the simulated Internet. We also calculated the number of network hops from a real host at Harvard to all the other hosts on the real Internet. The results are shown in Figure 4.2.

The relatively high correlation indicates that the data are similar; the y

Figure 4.1: **Geographical Host Locations.** Each potential client in our database is represented by a dot, plotted using the longitude and latitude information gathered for each host. Note the distinctive shape of the United States, as well as the presence of clients in Hawaii and Alaska.

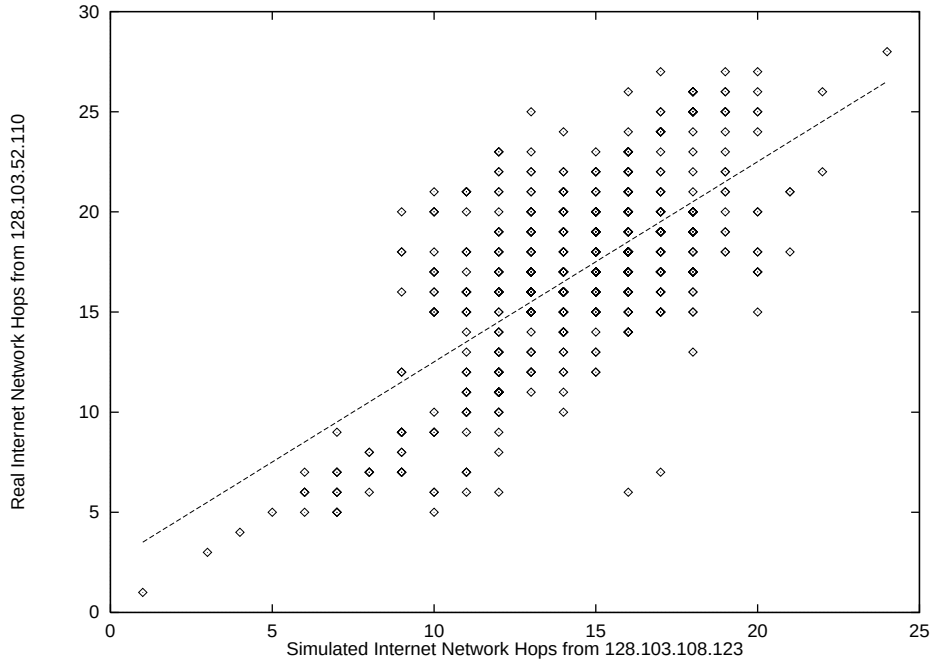


Figure 4.2: **Simulated Internet vs. Real Internet.** Here we have plotted network hops to the simulated hosts against networks hops to the real hosts. The regression fit correlation is $r^2 = 0.47$, the fitted line's equation is $y = 1.0x + 2.51$.

offset on the fitted line indicates that the simulated network hops are lower overall than the real network hops. The differences between the simulated and real networks stem from three causes. First, the simulated data were gathered over a year ago and many of the hosts from the original survey are no longer accessible. Out of the 1700 simulated hosts we were unable to contact 125 real hosts. Secondly, we did not have access to the actual machine at Harvard present in the simulated topology. We had to settle for running our traces from a Harvard machine that is one hop farther away from the backbone than the simulated machine. Thirdly, the real Internet does not always perform optimal routing: the path that a packet takes between two hosts is not always the shortest path. We can not replicate this routing exactly in our simulated Internet—our packets always traveled the shortest route. These differences are non-trivial, but will not skew our results since all of our caching schemes will use the same topology information.

Since our simulated Internet contains only 1700 hosts and our traces contain up to 50,000 hosts, we needed to create a mapping from real hosts to simulated hosts. Requests in a simulated run appear to be initiated from the mapped host, not from the original host. In computing this mapping we therefore try to maintain geographical information as much as possible. We first try to map a host onto a simulated host in the same subnetwork of the real host. For example, any requests from hosts on the 128.103 subnet will be mapped to a simulated host also on the 128.103 subnet if possible. If such a mapping is not possible then we try to map the host onto a randomly selected host in the same state. This is possible for all states except Vermont or Wyoming, because no hosts in the topology were located in either of those two states. For requests from those states we choose a mapping completely at random.

We did not initially maintain host mappings between simulation runs. Several of our early results showed excessive noise, however, and we realized that this randomization in the host mapping was skewing our results. We therefore set up the mapping such that once a host is mapped it remains mapped for all simulations. This was a valuable lesson because it showed us the sensitivity with which host-mapping affects network bandwidth consumption. We expect therefore that any parameter that affects server selection will affect bandwidth consumption.

4.2.2 Implementation Details

The simulator was written in C++ to simplify the implementation of its modules. Each module was written as a C++ class, and we took advantage of inheritance to simplify the substitution of cache-scheme dependent operations. As an example, the network always hands its messages off to a CHost object. Since the server and client classes inherit their interface from the host class, different types of hosts can be substituted into the network without having to change any network code.

This approach is also efficient because scheme-independent code need only be written once in the generic module, and then inherited modules can call it. This applies, for example, to the instrumentation built into all the host classes to monitor load. To reduce the simulator's memory requirements we used a disk-backed database to store information such as which clients are caching which documents, the simulated Internet topology, and the host mapping.

We have also built into the simulator extensive logging tools that were used for both debugging as well as gathering results. The debug logs enabled us to test the operation of the simulator by running with contrived test cases in order to verify push-cache operation.

4.3 Simulator Parameters

The remaining open issue is the parameterization of our simulator. Our algorithm relies on the server's ability to distribute objects to remote sites in a manner consistent with the request stream. That leaves several important decisions to make:

- When should a server push files to a push-cache server?
- To which push-cache server should a server push its files?
- How many files should be pushed at a time?
- How many servers are needed overall for effective push-caching?
- From which push-cache server should a client receive its files?

The first four questions are "Push Questions" as they describe the server's attempts to push data to other servers, and the last question is a "Pull Question" since it describes from where the client will pull information.

These questions are the topics of the following sections; we answer them in chapter 5.

4.3.1 When to Push

Deciding when to push is very important because of the great range of popularity experienced by servers on the Internet. A globally popular Web server that receives thousands of requests per hour should obviously have push-caching priority over a more obscure server that only receives a few hundred requests per day. To address this question we used an absolute `push-threshold` parameter.

Each server maintains a load factor based on how many requests it services per hour. Every time the server satisfies a request for a document it increments its load. Every half hour the server computes $load_{new} = load_{old} \times decay$ where $decay = 0.5$. We chose this formula based on the load formula used by Unix [22] in order to achieve some continuity across load levels. When a server exceeds the `push-threshold` it replicates its most popular files to the optimal location, resetting the load to zero in the process.

In this manner we arbitrate between popular servers and the less popular. The `push-threshold` will need to be reset from time to time to keep up with the growth of the Web; we envision that the same service that maintains a list of available servers will also maintain the current threshold value.

A related issue concerns stability; we want to prevent a rogue server from flooding caches with thousands of unpopular pages, denying access to legitimate servers. There is no way to deal with this problem directly without imposing a bottleneck at the point where access is controlled, but the problem is also self-correcting. The replacement policy on push-cache servers should be directly tied to the popularity of the pages cached such that servers replace the least-popular pages first. Pages pushed illegitimately that are not actually popular will therefore quickly be replaced by genuinely popular pages. Caches should also establish a cap on the number of pages they accept from any one server.

We expected that pushing pages frequently would be more efficient than pushing infrequently, but we also thought that there would be a point of diminishing returns. Pushing too frequently would inundate push-cache servers with requests and would be too sensitive to momentary abberations in request patterns.

4.3.2 Where to Push

The server must record information describing the source of Web requests in order to efficiently decide where to push documents. Originally we tried reducing the amount of information that the server needed to store by storing access history information at a coarse level. Our first try recorded the number of requests arriving from each state.

Using states, however, proved to be a particularly inefficient because network access does not fall along state lines. Consider the case of a file that is popular on both the east coast and the west coast. We found in early simulations that almost all of the west coast accesses will be from California, whereas the east coast accesses will be spread across several states including New York, Massachusetts, Rhode Island, Vermont, and Maryland. Figure 4.3 shows the state requests graphically for the NCSA access log analyzed above.

10,000 requests clustered in New England compares favorably to 9500 requests from California, yet due to arbitrary political distinctions the 10,000 requests from New England are divided among 8 states. California will dominate the decision of where to cache documents because its single large number will dominate the collection of much smaller numbers from New England.

The solution is not to make the grouping even more coarse; dividing the country into seven regions as suggested by Claffy and Braun [7] might help distribute server load but would not have a noticeable effect on network bandwidth.

Instead we record the number of requests that arrive from each individual subnetwork. This information can be used to calculate the optimal location of file replicas, but would overwhelm the server if a separate log were kept for each individual file. An analysis of the server logs showed that this was not necessary.

As shown in Section 3.1.1, a few files make up a large percentage of the requests to a given server. Since these are precisely the pages that are most efficiently replicated, we are most interested in recording the access history for these files. It turns out that the request pattern for the entire server closely parallels the request pattern for the most popular files, and therefore only the request patterns for the server as a whole must be kept.

As a rough measure of fit we computed the access history for each file by recording how many requests arrived from each state as well as for the server as a whole. We computed the correlation between the request pattern

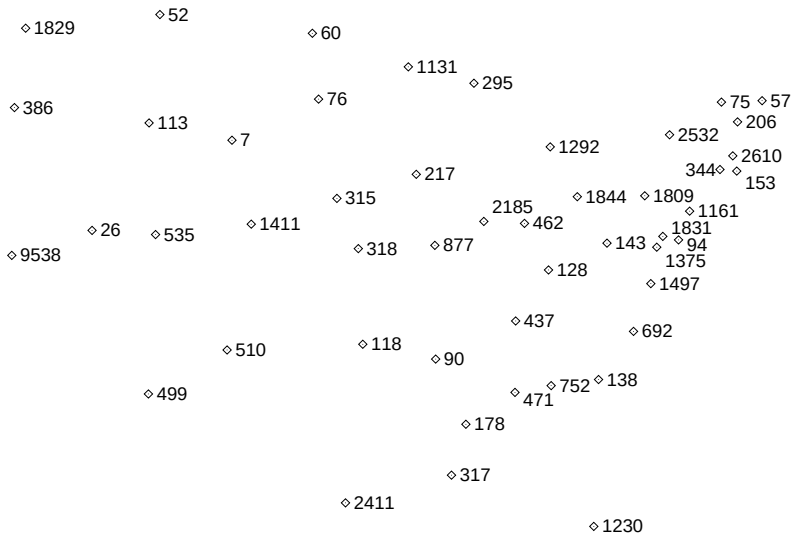


Figure 4.3: **Visualization of each state's requests to one of NCSA's servers.** Each state's label is placed over an arbitrary city located near the center of that state.

File Name	Number of requests	Percent of total	Correlation
NCSAMosaicHome.html	10296	15.77%	0.9943
mosaic.gif	7759	11.88%	0.9959
NetworkStartingPoints.html	2860	4.38%	0.9942
whats-new.html	1957	3.0%	0.9921
WinMosaic/HomePage.html	1466	2.25%	0.9550
MetaIndex.html	1410	2.16%	0.9790
NCSAWN.gif	1323	2.03%	0.9867
NCSAWN-MOS.gif	1319	2.02%	0.9880
skill.gif	1301	1.99%	0.9885
NCSAWN-GNN.gif	1298	1.99%	0.9862

Table 4.1: **Correlation between the location of requests for the 10 most popular files on an NCSA server and the location of requests for the entire server.**

for the top 10 most popular files from the NCSA server trace and for the server as a whole. These correlations are listed in table 4.1. The average correlation in this table is 0.98; these extremely high correlations indicate that it is sufficient to record information only about the entire server as a whole, and not each file individually.

Our simulated server maintains a list of the subnets from which requests have been received along with the number of requests received from each. This list is used to decide where pages get replicated. When a pushing decision is made the list is cleared. It should never grow above several thousand networks since that would indicate that pages should be sent to remote servers. Old entries should also be removed periodically to avoid using stale information in making push-caching decisions.

To actually make the replication decision, the server uses a modified *friends-of-friends* algorithm (see section 3.2). We "seed" the algorithm with a selection of existing remote servers, one remote server per cluster. The server then iterates through the access history list just described, adding each subnet's number of requests to the cluster nearest to that subnet. Once this is complete the server then pushes its most popular files to the push-cache server in the cluster responsible for the greatest number of requests.

The `caching-strategy` parameter helps decide what defines "nearest" .

We experimented with three different methods: by network hops, by miles, and by random choice. According to the hops metric, the nearest server is the fewest network hops away. According to the miles metric, the nearest server is the fewest miles away. And according to the random metric, the nearest server is chosen randomly.

We expected that making the decision randomly would have the worst performance, and that choosing optimally by using the hops to make the choice would have the best performance. We had hoped that using miles to decide would approximate the performance of using hops, because currently miles is a feasible metric on the real Internet while hops is not.

4.3.3 How much to push?

Server-initiated caching is most efficient when overhead operations, such as pushing documents, are minimized. Servers should therefore push multiple pages. We experimented with the the number of the most popular files pushed at a time, the `pages-to-push`[1] parameter. We had expected that pushing many pages at a time would be more efficient than pushing a few pages at a time because it would amortize the overhead of push-caching over more pages.

4.3.4 Locating Caches

It is not enough to place a replicated copy of several files in the optimal location; local clients must also learn about the location of these cached files in order to use them. Other groups are working on sophisticated ways to handle local resource discovery [6, 19]. This question is outside the domain of this work, so we used the simplest method suggested by Blaze and discussed in section 2.1.1. Clients contact a file's primary host on first request. The server may then send the client a redirect message instructing it to use a different cache instead, along with a list of other files currently cached at that other site. The client caches this list of redirects, and uses it to decide which server to contact for files in which it is interested. Using a more efficient location scheme will only improve the performance of push-caching.

To decide which push-cache is closest to the client our simulator uses the same three metrics listed above: miles, network hops, and random selection. The parameter `lookup-strategy` decides which policy is in effect; for every request it receives the server checks to see if one of the caches to which it has

[1] We use bold face to denote a simulator parameter

pushed that document is closer to the client than the server itself. If not, it simply returns the document. If so, it sends the client a redirect message for each of its documents currently cached at that server. This policy is designed to minimize the number of server queries necessary, amortizing the cost of querying a distant server over the many local requests that are thereby made possible.

On average, clients request several documents (4-20) from a server. If the server acts preemptively, sending the client redirects for all of its documents that are currently being cached it should be able to avoid multiple queries. Given that the average file size on the traces we examined is 2000 bytes and that a redirect consists of a file address (average length 30 bytes) and a machine name (4 bytes), we can send approximately 60 redirects for the price of one file. Additionally, if push-caching enables us to halve the expected latency of a transaction, then transferring only two documents from a closer push-cache server makes up for querying the original server.

On the client side, caching redirects takes much less space than caching full documents. The client always tries a cached redirect if it has one since the push-cache server manages consistency, not the client. The client does not therefore have to worry about stale documents, only stale redirects. Even these are not a problem, however, since a push-cache server that is no longer storing the document will return an error message. In this case the client should simply contact the original server. Since the client should have been redirected to a nearby push-cache server the invalid message will be returned quickly.

As an implementation note, many sites are currently using proxy servers. The client-side of this algorithm is implemented in the proxy server more easily than it is implemented in the client, since the proxy server is assumed to have the space to cache many tens of thousands of redirects as well as thousands of documents.

We expected here as well that using hops would be most efficient, followed closely by miles, and more distantly by random selection.

4.3.5 How many servers do we need?

For push-caching to be effective, there must obviously be push-cache servers available onto which objects may be pushed. We experimented with the number of servers available, the `server-number` parameter. This parameter must be distinguished from one of the metrics that we will use in the next chapter to measure push-cache performance, the number of active push-

cache servers. We did not expect all of the available push-cache servers to be used, because once a set of objects was sufficiently replicated we did not expect the load on any one server to exceed the push-cache threshold.

We do, however, expect network traffic to diminish as more servers are made available, because the larger the pool of available servers, the more likely that a server will be available where it can optimally reduce traffic.

Chapter 5

Simulator Results

We used our simulator to explore how the five parameters discussed in section 4.3 affect the performance of push-caching. These parameters are summarized in table 5.1. The results of these simulations are presented in sections 5.1 through 5.4. We also used our simulator to compare push-caching to client-caching, where each client caches a small set of popular documents, and proxy-caching, discussed in section 2.1.3. The results of these simulations are presented in section 5.5 and 5.6. We conclude this chapter with a discussion of push-caching scalability.

5.1 `Caching-strategy` and `Lookup-strategy`

We began by examining the two sets of strategies: `caching-strategy` and `lookup-strategy`. Each set has three members; there are nine combinations altogether. For each combination we simulated several traces and averaged the results together in order to limit the introduction of random variation. We used four different metrics to evaluate performance: total network bandwidth consumed, primary host traffic, average push-cache server traffic, and the number of push-cache servers actually used.

We set the other parameters for these simulations according to our intuition, described in the previous chapter. We set the `push-threshold` to 20, the `pages-to-push` to 10, and `server-number` to 50. We took our traces from the NCSA server trace, pulling out three sets of results each with 20% of the total NCSA events. The results are summarized in tables 5.2 through 5.4.

We had expected that using hops as a distance-metric would be more

Parameter	Description	Setting range
`caching-strategy`	Metric used by server to decide where to push.	$\{random, miles, hops\}$
`lookup-strategy`	Metric used by client to determine nearest cache.	$\{random, miles, hops\}$
`push-threshold`	Load threshold at which server pushes pages.	$[0...100,000]$
`pages-to-push`	How many pages the server pushes at a time.	$[0...100]$
`server-number`	How many push-servers are available.	$[0...100]$

Table 5.1: **Summary of the parameters explored by our experiments**

Push Strategy / Pull Strategy	hops	miles	random
hops	75.54%	77.40%	89.70%
miles	87.01%	87.62%	102.99%
random	105.54%	102.17%	119.53%

Table 5.2: **Effects of pulling and pushing strategies on** *network bandwidth.* 100% = bandwidth consumed with no caching at all. Notice that pull strategy has a greater effect on network bandwidth than the pushing strategy, that random pulling selection uses more bandwidth than no caching, and that we were correct in our assumption that hops are generally more effective at reducing bandwidth consumption than miles and miles are more effective than random selection.

Push Strategy / Pull Strategy	hops	miles	random
hops	40.46%	42.69%	41.26%
miles	44.81%	46.49%	45.33%
random	39.96%	40.15%	39.39%

Table 5.3: **Effects of pulling and pushing strategies on** *primary server traffic*. 100% = primary server's traffic with no caching at all. Notice that once again the pull strategy has a greater effect on server load than the pushing strategy, although here the differences are less pronounced. Notice also that random pulling is very effective at reducing load. This makes sense, because pulling randomly distributes the load evenly over all available push-cache servers.

Push Strategy / Pull Strategy	hops	miles	random
hops	46	34	50
miles	44	33	48
random	47	31	50

Table 5.4: **Effects of pulling and pushing strategies on the** *number of active servers*. For these simulations there were 50 servers available. Notice that random pushing can saturate all available servers, and that pushing strategy exerts more influence over the server number than pulling strategy.

effective at reducing network bandwidth consumption than using miles, and that using miles as a distance-metric would in turn be more effective at reducing bandwidth consumption than choosing randomly. Our simulation confirmed these hypotheses.

Our simulation denied, however, our hypothesis that random pulling and random pushing do not use significantly more bandwidth than no caching at all. Initially we found that random pulling was using significantly more bandwidth than directed pulling. We examined our simulator to determine if this was a robust result or a bug, and found that it was neither. With random selection a client can fall into a long redirect loop where it may be redirected to several different hosts before finally retrieving a file. To eliminate this possibility we added a redirect counter to each request that prevents it from being redirected to more than 3 servers. This counter eliminated the cost of redirection loops and reduced the bandwidth costs associated with random selection.

Random pulling still consumes more bandwidth than no caching, however, even after removing the redirect loops. This is because of the overhead that push-caching creates through redirect messages and pushing documents. We purposely used a low `push-threshold` in this section; in the next section we will show that a larger threshold is more reasonable, and the overhead will therefore decrease somewhat.

One result we had not expected was that `lookup-strategy` has a greater effect on bandwidth consumption than `caching-strategy`. This effect is seen in the tight clustering of the values within each pulling strategy, and the wide spread of values within each pushing strategy, and indicates that discovering nearby resources efficiently is very important. This result makes intuitive sense; if object replicas are distributed throughout the network, clients will save network bandwidth by accessing the nearest replica. Servers influence the savings by placing replicas in the optimal locations, but random distribution is still effective because it distributes replicas evenly.

It is interesting to note that while random pulling uses network bandwidth inefficiently, it does a very good job of distributing server load. This is explained by the fact that random pulling distributes client requests evenly over all the available servers, dividing the primary host's load by n, where n is the number of push-caching servers in use. The primary host load did not decline to $1/n$ because it only replicates its most popular files. Requests to the rest of its files must still be satisfied by the primary host.

Hops are similarly effective, but for a different reason. The effectiveness of using hops as a pulling strategy to reduce primary server load is related to

the distribution of client requests. Since clients select from the nearest push-cache when they are using the hops metric, if client requests are distributed evenly across the network, push-caches are distributed evenly across the network, and objects are replicated evenly across caches, then primary server load will again be divided by n.

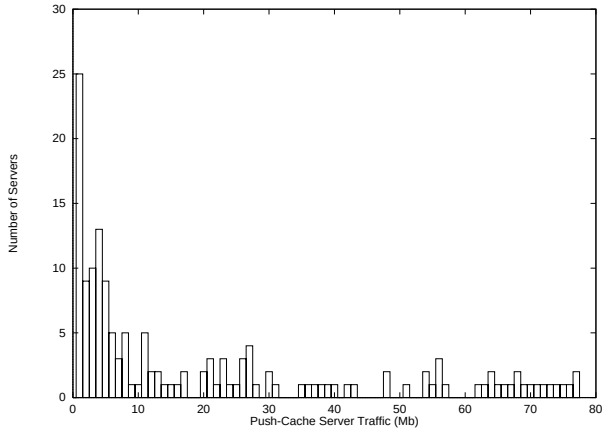
For globally popular servers our assumptions about even client distribution will hold, and therefore a hops-based strategy will be effective at distributing load. For servers with less global popularity, random pulling will be more effective at distributing load, although less effective at saving bandwidth.

We see the exact server traffic distributions by plotting server traffic histograms. Figure 5.1 presents the server load distribution when using both hops and random strategies for pulling and pushing.

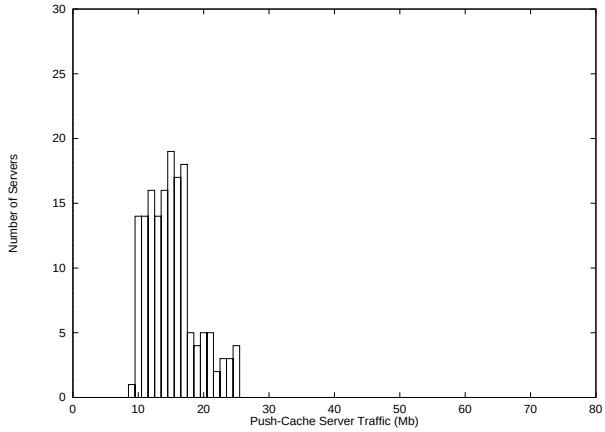
The histograms make sense; random pulling selects among the servers equally, therefore servers should see an equivalent load. The apparent uneven distribution in (b) results from starting our simulations cold, with no documents being cached anywhere. Those servers that receive documents early in the simulation will see more traffic than those that receive documents later in the simulation. This cold start also plays a factor in (a), but so does server location. A server located in a very active cluster will see more traffic than a server located in a less active cluster. These effects are self correcting, however, since busy servers will replicate files to reduce their load.

In table 5.4 we see that random-pushing and random-pulling saturated all available servers; this is not a fluke, but a robust effect resulting from a relatively low `push-threshold` and random selection. With a low threshold, a server only needs to receive a few requests before it decides to push its files. With random pushing, these files will eventually be distributed over a large number of servers. With random pulling, however, each of these servers will begin receiving requests for their files and soon each of these servers will decide to push their own files. Within a short time all servers have received the file. Saturation of this sort is not desirable because it does not make efficient use of cache space, distributing objects so thinly that they receive few requests. The solution is to raise the threshold and avoid random pulling.

Our conclusion from this section is that hops-based selection is optimal for both pulling and pushing strategies for globally popular servers. Unfortunately, as we discussed in chapter 3, it is not currently possible to calculate hops on the Internet. Nevertheless, we are sufficiently satisfied



(a) Hops Based Selection



(b) Random Selection

Figure 5.1: **Effect of pulling and pushing strategies on server traffic distribution.** (a) displays the distribution when using hops to select where to push and pull, and (b) displays the distribution when randomly selecting where to push and pull. Notice that random selection distributes load fairly evenly across servers, whereas the hop-based selection leads to a skewed distribution where most hosts see little traffic and several hosts see significantly more traffic.

with the performance of miles-based prediction to recommend it in lieu of hops-based selection. We do not recommend random-pulling, but are interested in exploring random-pushing further, since random-pushing would eliminate the requirement that the server store geographic access information, yet still returns acceptable performance when coupled with hops or miles based pulling.

Throughout the rest of this chapter we will use hops-based pulling and pushing while exploring other parameters, although we will return to miles-based pulling and pushing for comparing push-caching to client-caching.

5.2 Push-threshold

To evaluate the impact of `push-threshold` on push-caching, we created an artificial trace that consisted of all the requests to a single item, the most popular item, from the NCSA trace. This reduced the number of events by a factor of 5, and also isolated the effect of `push-threshold`. We set `server-number` = 50, `pages-to-push` = 10, `lookup-strategy` = *hops*, and we simulated for `caching-strategy` = *hops*, *random*, and *miles*. The results are shown in Figures 5.2 and 5.3.

In these graphs we have averaged together the results of the three caching strategies in an effort to remove noise and isolate the effects of `push-threshold`. Ideally we would have performed these simulations many times with different push-cache servers and different traces as well, but we did not have sufficient time to perform these multiple simulations.

A low threshold results in the most aggressive push-caching; as the threshold rises the effects of push-caching diminish. Setting the threshold low results in low primary host traffic and low average server traffic. The price is the number of servers needed, since total server load is conserved.

The graphs indicate that a relatively low setting for the `push-threshold` is optimal, although we do not want to set it too low. The lower the setting, the more frequently that the primary server must push its pages and this incurs a fixed cost on the server that cannot be shown on these graphs. A threshold of 0, for example, results in the server pushing its pages every time it receives a single request. We want to therefore set the threshold to the highest setting that still results in excellent performance. We chose to set the `push-threshold` to 100 by judging the rate of change of the bandwidth graph. Between 0 and 100 the rate of change of network bandwidth is relatively low. We lose little performance by raising the threshold to 100. The

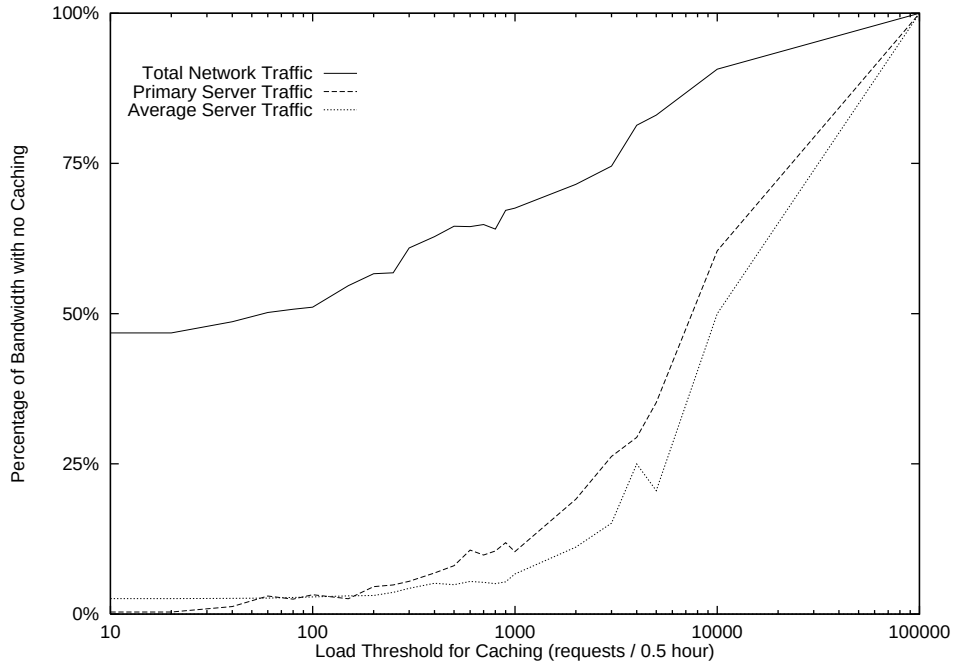


Figure 5.2: **Effects of `push-threshold` on total network traffic, primary server traffic, and average server traffic.** Note the use of a log-scale on the x-axis, and that as `push-threshold` approaches infinity the effects of push-caching diminish. 100% represents the value that would be produced with no caching, so 100,000 is therefore sufficiently large enough for there to be no push-caching. Notice that as the number of push-cache servers in use drops (Figure 5.3) the average server traffic rises. Notice also that the primary server traffic is different than the average server traffic; this means that load is distributed among the push-cache servers unequally.

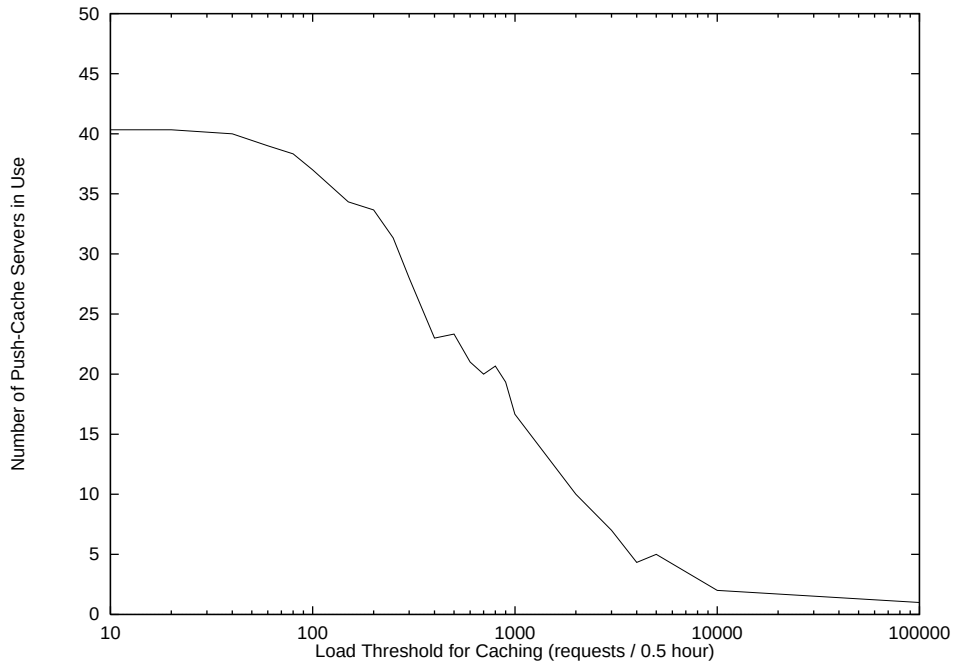


Figure 5.3: **Effects of `push-threshold` on the number of active push-cache servers.** Notice that with no caching there is exactly one push-cache server, the primary host.

rate of change increases, however, at 100 and so we begin to lose relatively more performance as the threshold continues to rise past 100.

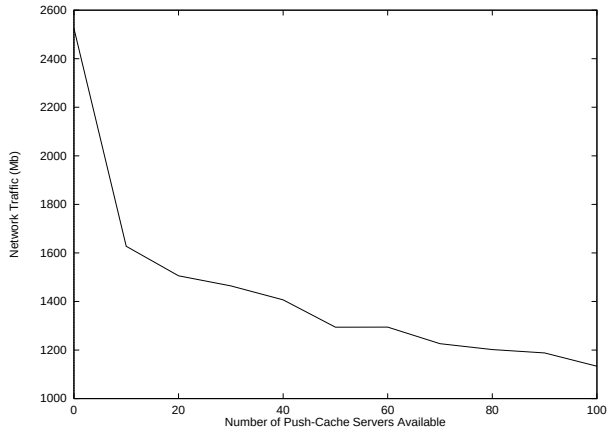
5.3 Server-number

We next examined the effect that changing the number of available push-cache servers had on push-caching. We used the same trace from the previous section, and once again we averaged together the results from simulating with the three different caching-strategies. We set `push-threshold` = 100, `pages-to-push` = 10, and `lookup-strategy` = *hops*. The results are shown in Figures 5.4 and 5.5.

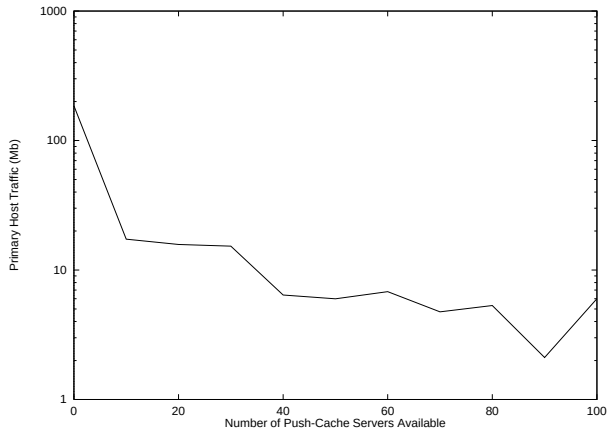
The simulation results are unremarkable, and confirm our hypothesis; the more servers available, the lower the bandwidth, but that not all the servers available would be used. The discontinuity on the right of the primary host traffic graph is interesting; it demonstrates push-caching's sensitivity to object distribution. The dip was caused by the simulation run that used randomly selected pushing and the hops metric to pull. The primary server randomly pushed several objects to another push-cache server early in the simulation. That push-cache server was closer to most other clients than the primary host, and therefore most client requests originally directed to the primary host were subsequently redirected to that secondary host. With more time available we would have averaged together additional runs to reduce such inconsistencies.

Figure 5.5 demonstrates that network traffic is reduced as more servers are made available, even as the percentage of servers in use declines. This is because with more servers available the distance-based metrics can do a better job of selecting the optimal cache server given the current access pattern. This observation emphasizes the fact that push-caching is very sensitive to network topology. It is also interesting to note that average server traffic remains essentially constant beyond 50 servers or so.

We conclude that increasing the number of servers improves the performance of push-caching, but caution that increasing the number of available servers also decreases the performance of the algorithm used to select where to replicate an object. Beyond several hundred servers or so it becomes infeasible to exhaustively search through them to determine the optimal server, since the computation time is in $O(m \times n)$ where m is the number of available servers, and n is the number of clients that have made requests. A constant factor that should not be ignored either is the cost of calculat-

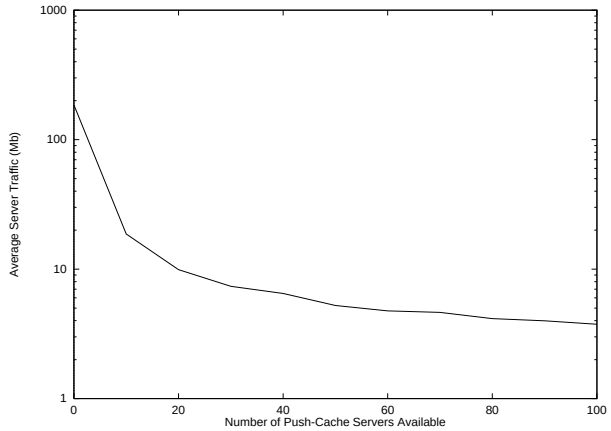


(a) Network Traffic

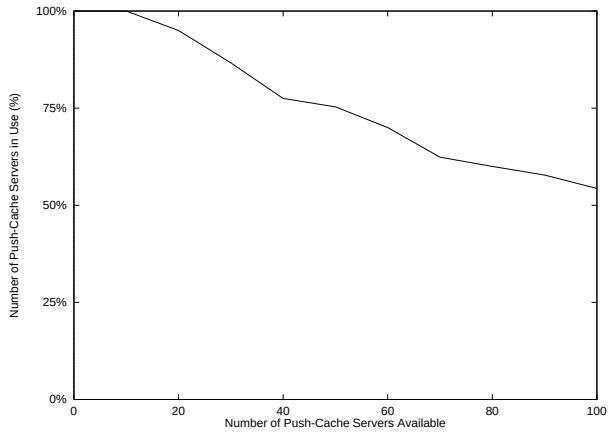


(b) Primary Host Traffic

Figure 5.4: **The effect of `server-number` on network traffic and primary host traffic.** Note the discontinuity at the right of the server traffic graph; see the text for an explanation.



(a) Average Server Traffic



(b) Number of Active Push-Cache Servers

Figure 5.5: **The effect of `server-number` on average server traffic and active push-cache servers.** Notice that as the number of available servers increases the percentage of those servers in use decreases.

ing the distance between two clients, since it will computed up to $m \times n$ times. We will use `server-number` = 50 for the rest of the simulations in this section as a tradeoff between performance and computability.

As an implementation aside, this computation should obviously not be allowed to delay the handling of requests by the push-cache server and should be handled by a separate thread or process.

An area of future study is an examination of the various heuristics that could be used to narrow the set of servers under consideration. This heuristic would be used by the registry (see section 1.1) to reduce the selection of available servers offered to the server about to replicate its objects. We predict that simple random selection will be most effective.

5.4 Pages-to-Push

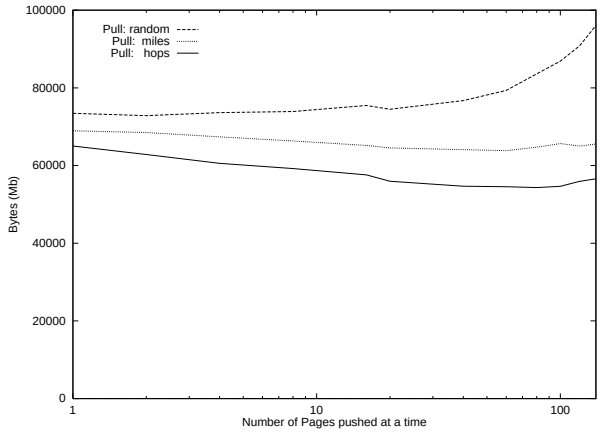
The next parameter we varied was the number of pages to push at a time. The results are shown in Figures 5.6 and 5.7.

Our results did not match our intuition. As we increased the number of pages pushed at a time we noticed the expected initial decrease in bandwidth. However, for all but the random pulling strategy, the bandwidth consumption remained essentially flat as we continued to increase the number of pages pushed.

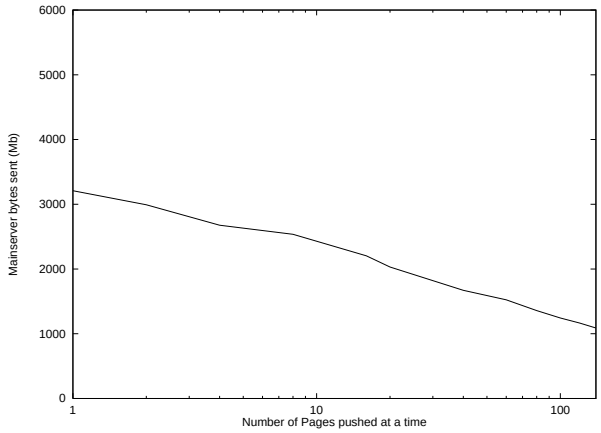
The linear increase in bandwidth for random server selection as the number of pages increases was the clue that helped us decipher these results. There are actually two effects that arise from increasing the number of pages: On one hand, increasing the number of pages increases the overhead of push-caching. On the other hand, increasing the number of pages improves push-caching's performance because more objects are available to clients from nearby servers. These two effects cancel each other out in the case of miles and hops server selection, but since there is effectively no push-caching in the case of the random server selection we see only the rise in bandwidth due to increased overhead costs, and none of the decrease due to more efficient caching.

The fact that the number of servers used remains essentially constant regardless of the number of pages pushed is not surprising, because the number of servers used is dependent on client access patterns, not the efficiency of the caching once the clients locate the servers.

So far this data indicates that `pages-to-push` should be set as high as possible, because there are no apparent side effects from a high `pages-to-push`

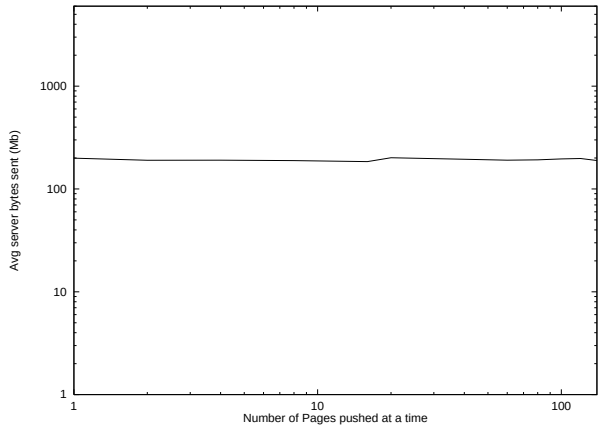


(a) Network Traffic

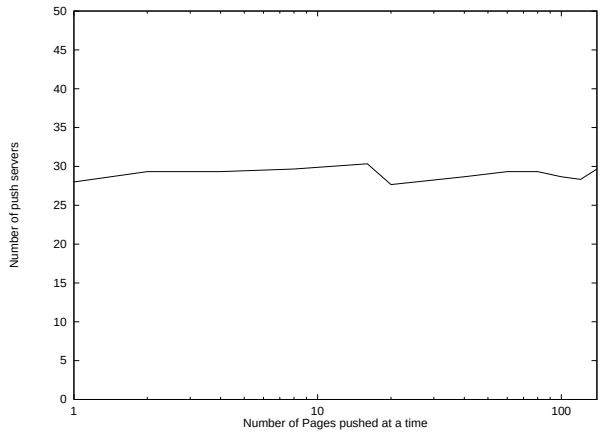


(b) Primary Host Traffic

Figure 5.6: **The effect of `pages-to-push` on network traffic and primary host traffic.** Line-type denotes the pulling strategy in (a), strategies are averaged together in (b). The server trace shown in these graphs is the NCSA trace. Notice that the network traffic created by random pulling increases as the number of pages pushed at a time increases.



(a) Average Server Traffic



(b) Number of Active Push-Cache Servers

Figure 5.7: **The effect of `pages-to-push` on average server traffic and active push-cache servers.** The server trace shown here is the NCSA trace. Notice that the average server traffic and the active server number remain essentially constant.

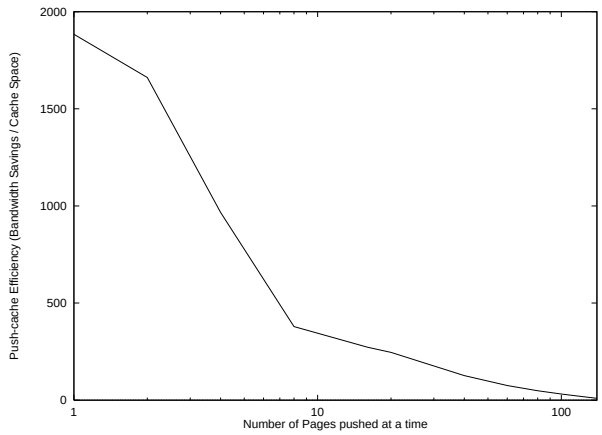


Figure 5.8: **Effect of `pages-to-push` on push-cache efficiency.** Note the use of a log-scale on the x-axis to enhance readability. As the number of pages pushed increases, the efficiency decreases.

parameter. This is because we have not yet considered the amount of cache space required at the push-cache servers in order to store the replicated pages. We can compute the efficiency of push-caching as `bandwidth saved/cache space required`, as discussed in section 2.4.1; Figure 5.8 displays this efficiency.

We see from this graph the cost of increasing the `pages-to-push` parameter: a declining efficiency. We suggest therefore that `pages-to-push` be set at 10. Performance is not improved significantly by setting it any higher, but the efficiency declines precipitously.

5.5 Client-initiated caching vs. Push-caching

To conclude our study of push-caching, we compared push-caching to traditional client-caching. We simulated client-caching by assuming that once a client requests a given document that the same client will never again request that document. A side effect of this approach is that caches are assumed to be infinitely large, an unrealistic assumption because client caches are frequently small, but one which simplifies the simulation significantly because we do not have to worry about cache invalidation policies. Re-

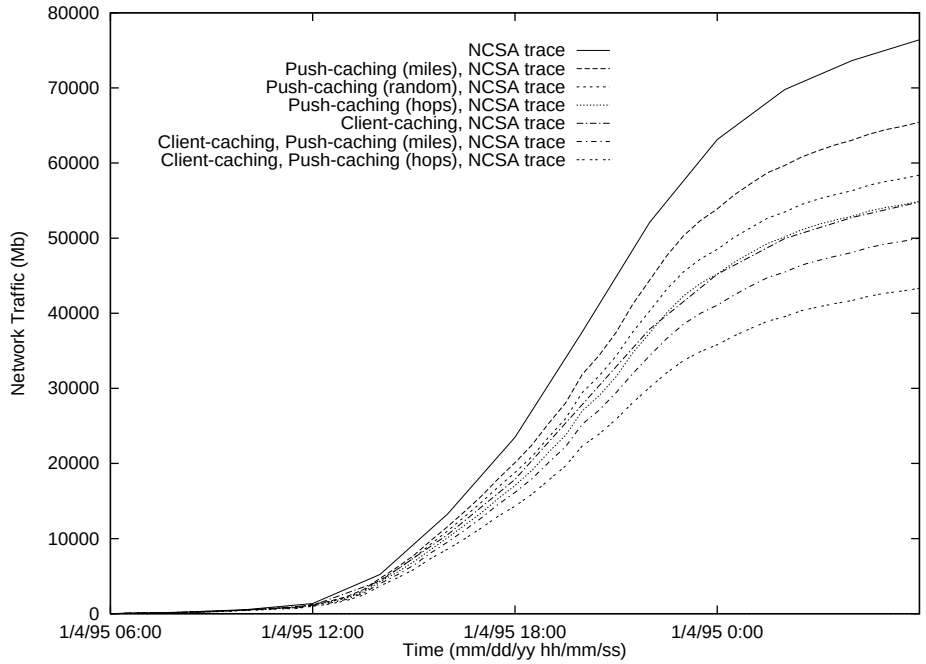


Figure 5.9: **A Comparison of client caching to push-caching on the NCSA trace.** Note that hops-based push-caching and client-caching have equivalent performance, and that the combination of hops-based push-caching and client-caching is the most effective at reducing bandwidth.

sults for client-caching will therefore be somewhat optimistic. We used the values for server-initiated caching derived from previous sections, setting the `caching-strategy` to *miles*, and setting the `pull-strategy` to *miles*, *hops*, and in the case of NCSA, *random* as well. The results are shown in Figures 5.9 through 5.12.

Caching was most effective on the two most popular traces, NCSA and FAS, thus affirming the hypothesis of section 3.1.1. Client-caching and optimal push-caching used the same amount of bandwidth, but they clearly were affecting different sets of objects because their combination proved most effective. Table 5.5 displays more detailed statistics for the most popular trace, the NCSA trace. Notice that random pushing, while not as effective at reducing network traffic as hops pushing, was nevertheless more effective than using miles to push.

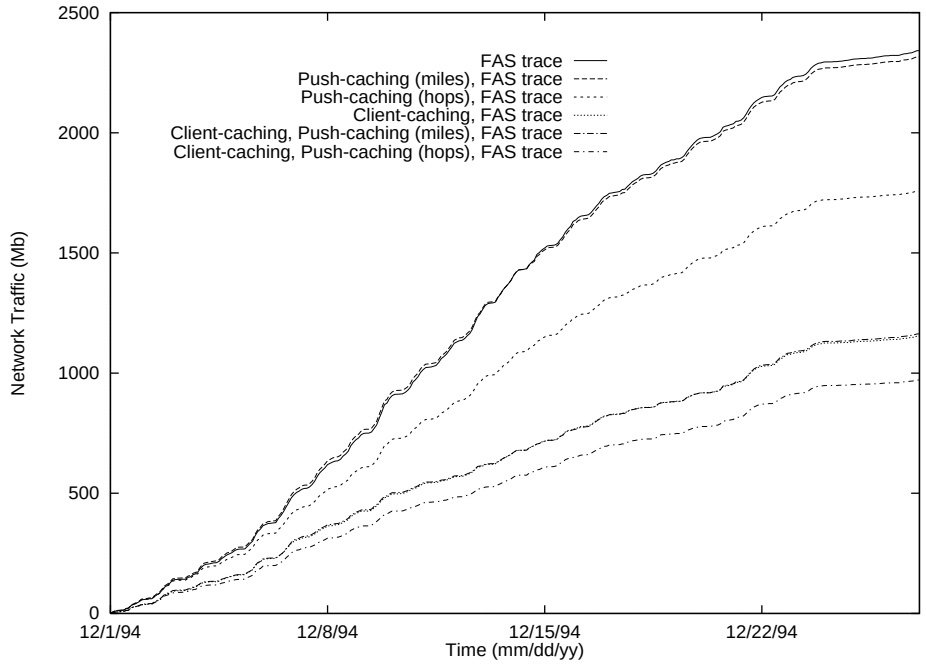


Figure 5.10: **A Comparison of client caching to push-caching on the FAS trace.** Note that client-caching is extremely effective because FAS is a locally popular server. Most of its requests come from the same machines at Harvard including public, shared workstations.

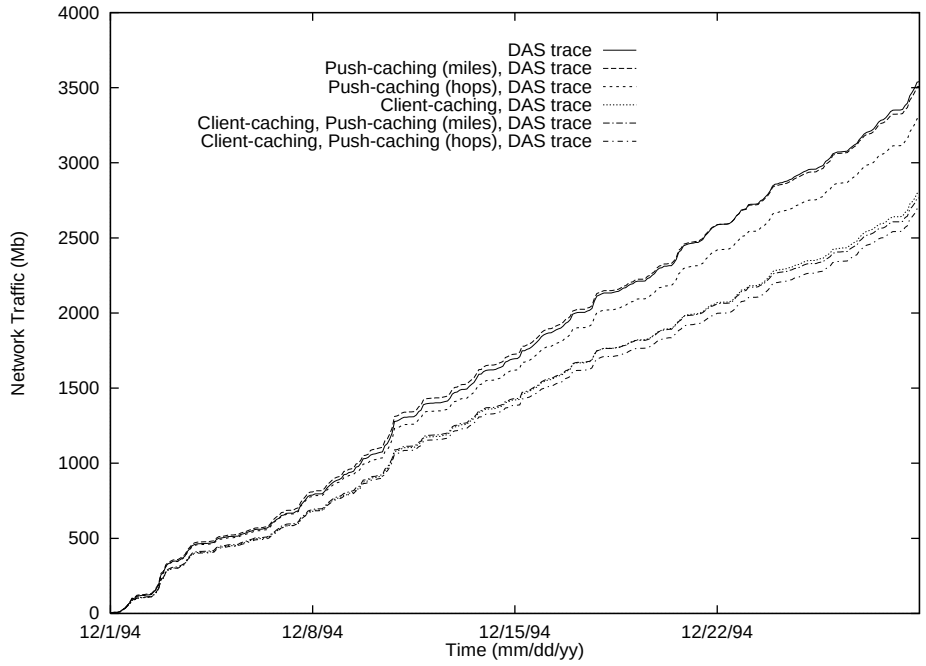


Figure 5.11: **A Comparison of client caching to push-caching on the DAS trace.** Push-caching is particularly ineffective for this trace; Figure 3.1 shows why. This server is the most diverse in that there is no small group of files accounting for the majority of requests, so there is no core group for push-caching to distribute.

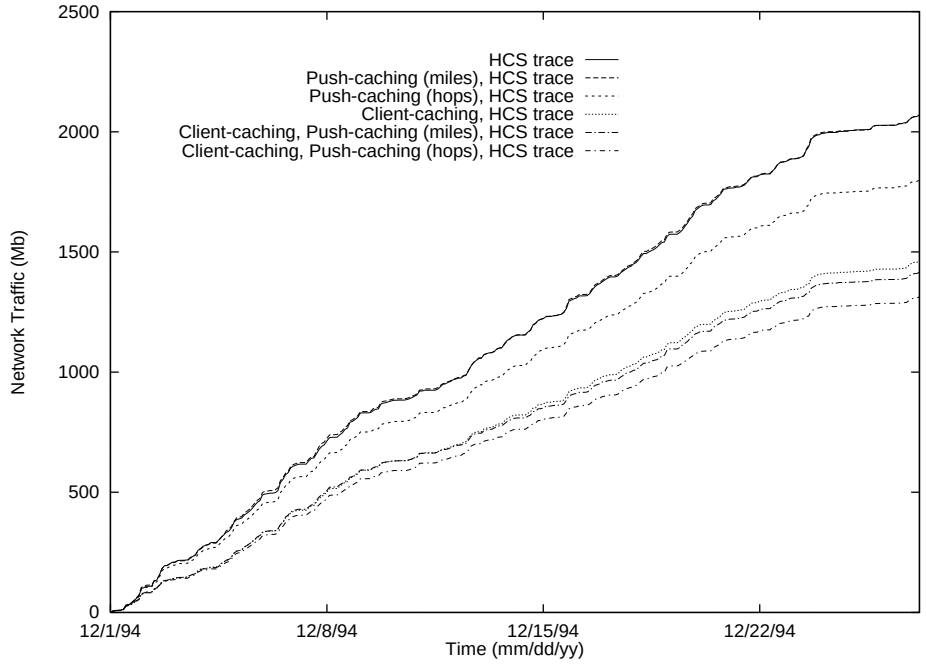


Figure 5.12: **A Comparison of client caching to push-caching on the HCS trace.** The local popularity of this server accounts for the success of client-caching relative to push-caching.

Caching Method	Net traffic	Primary Server Traffic	Average server Traffic	Active Servers
None	100%	100%	100%	1
Push-Caching (miles)	85.6%	46.8%	2.8%	35
Push-Caching (random)	76.4%	40.9%	1.9%	51
Push-Caching (hops)	71.8%	39.6%	2.1%	47
Client Caching	71.7%	71.6%	71.6%	1
Client & Push-Caching (miles)	65.4%	40.2%	2.1%	34
Client & Push-Caching (hops)	56.7%	33.7%	1.5%	47

Table 5.5: **Detailed examination of client-initiated caching versus server-initiated caching.** The addition of server-initiated caching helps decrease network bandwidth consumption, but it also reduces the primary server load considerably. Note that we have included the primary server in the average server traffic calculation.

These simulations clearly show a need for all Web browsers to implement client caching, since the performance improvement is so dramatic, but they also show the advantages of push-caching in reducing the primary server load and network bandwidth consumption.

5.6 Proxy-caching vs. Push-caching

We close the results section by comparing proxy-caching to push-caching. As we saw in section 2.1.3, one proxy vendor claimed that large proxy caches can have hit rates of up to 65%. We wanted to evaluate their performance, expecting proxy-caches to perform even more efficiently than the simple client-caching described above. Proxy-caches are also a rudimentary form of hierarchical caching since multiple computers share one cache. By comparing proxy-caching to push-caching we are also therefore comparing simple hierarchical caching to push-caching.

As we discussed in section 4.2.1, our simulator currently maps hosts from the trace data onto 1700 simulated hosts. In the case of the NCSA trace data, for example, this involves mapping 50,000 hosts onto 1700. This means that on average there are 29 hosts mapped to each one of the simulated hosts. If we allow these mapped hosts to satisfy their requests from documents

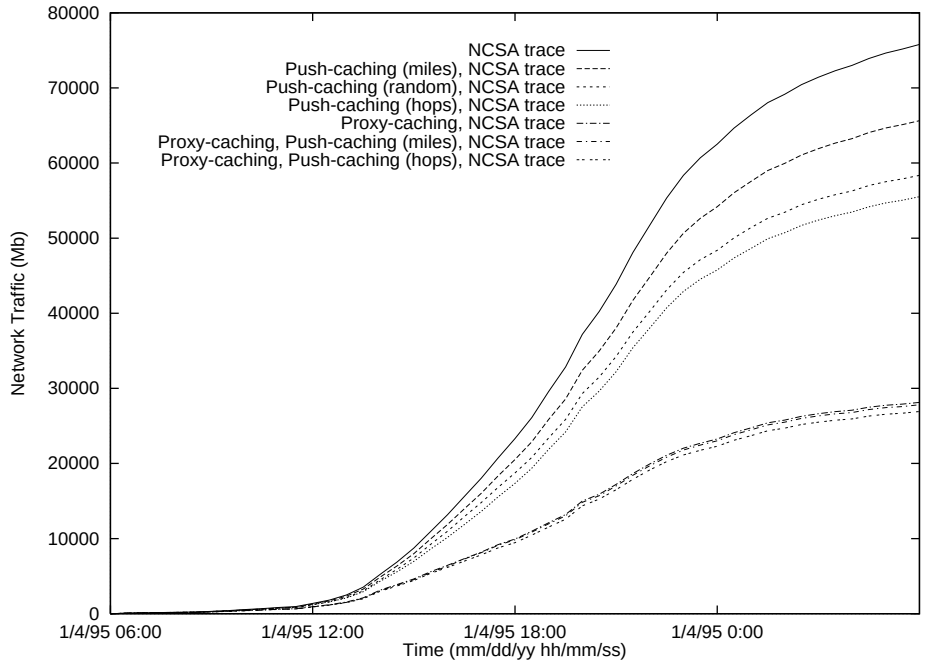


Figure 5.13: **Proxy-caching Vs. Push-caching.** The NCSA trace was used in this simulation. Notice that proxy-caching saves far more bandwidth than push-caching alone, and that adding push-caching to proxy-caching does not significantly increase performance.

cached in the simulated host then we are simulating the effects of a Web proxy.

This is not a completely fair comparison; such a simulation is biased in favor of proxy-caching. Our trace data is server-centric; it was taken from a single server. To assume that all the hosts connected to a given proxy-cache are accessing data from the same server favors the proxy cache, and therefore the results from this section should not be considered realistic, but rather optimal for proxy-caching. They are useful as a benchmark against which to consider push-caching because in the real-world push-caching will only do better when compared against proxy-caching. This is because in the real world clients would be asking proxy-caches for documents from many servers, not just a single server, and not all servers will be as globally popular as the NCSA server.

Figure 5.13 displays a simulation of the NCSA server trace. As expected, optimal proxy-caching is extremely efficient in terms of bandwidth, saving 60% over the original bandwidth consumption. Miles-based push-caching saves only 14%, and the optimal push-caching case, hops-based push-caching, only 26% of the bandwidth. Combining push-caching with proxy-caching only adds at most an additional 2% bandwidth reduction over proxy-caching alone.

Table 5.6 presents more detail comparing the two traces. The efficiency column is the ratio of bandwidth savings over cache space required; the metric discussed in section 2.4.1 and used above. Braun and Claffy had simulated geographical caching and found at most an efficiency of 7 with their simulations. We found proxy-caching to be more efficient, with an efficiency of 21. However, push-caching is extremely efficient, with an order of magnitude increase over proxy-caching for similar server load savings. These efficiency calculations do not take into account the primary host which stores 200 times more objects than the average cache.

These results are easily explained: proxy-caching achieves its bandwidth savings by caching a copy of every document requested through the proxy regardless of whether the file is popular or not. Push-caching is far more selective, only replicating those files that are known to be popular. Since these files make up the majority of the requests to the primary host, push-caching is able to distribute the server's load far more efficiently than proxy-caching. One possible way to make proxy-caching more efficient would be to attach a popularity rating to each file distributed by a server. Proxy-caches can use this information to make more efficient use of their cache space.

5.7 Push-Caching Scalability

We have shown that push-caching is an efficient way to replicate documents across the Internet in order to reduce server load and network bandwidth. The last question we must address is whether push-caching scales to handle the millions of servers that will soon be using the Web. Push-caching depends on servers voluntarily accepting replicated documents, but no server will accept replicated documents if accepting them leads to a higher overall server load.

An analysis of the NCSA push-caching simulations shows that this is not the case. As we saw in Figure 5.9, the primary host's traffic was reduced from 5563 Mb to 2604 Mb when using miles to predict distance, for a savings

Caching Method	Band-width	Server Load	Total Cache Space (Mb)	Efficiency (savings / cache)
None	100%	100%	0	-
Push-Caching (miles)	86%	51%	8	1268
Push-Caching (hops)	74%	41%	27	746
Proxy-Caching	37%	36%	2219	21
Proxy & Push-Caching (miles)	36%	31%	2235	21
Proxy & Push-Caching (hops)	35%	29%	2256	21

Table 5.6: **Detailed examination of proxy-caching versus push-caching for the NCSA server trace.** Push-Caching is clearly far more efficient than client-based caching for similar savings in primary host load. Proxy-caching, however, saves significantly more network bandwidth than push-caching in return for an order of magnitude decrease in efficiency.

of 2959 Mb. Data was pushed to 34 servers, and the average traffic on each server was 87 Mb. If we assume that every server that replicates pages is also willing to accept replicated pages, and if we assume that every server that replicates pages is as popular as the NCSA servers, and if we assume that servers are selected uniformly, then since each server will replicate to 34 other servers, each server will also accept pages from 34 other servers. We may therefore calculate the increased load to be $34 \times 87 = 2958$, which is the same amount by which each server's load was decreased initially.

Push-caching has therefore decreased the amount of network traffic without significantly affecting each primary server's load. If only primary servers could store replicated files then push-caching would be of dubious value in reducing server load. The virtue of push-caching, however, is that it is very easy to add additional servers. Proxy-servers, for example, are ideal candidates for accepting replicated objects because they are already running web caching software, frequently are attached to large disks, and are usually not hidden behind firewalls. Push-caching can distribute the load from overloaded primary servers onto proxy-servers and other servers without imposing an unacceptable load because all servers caching replicated objects may refuse additional objects at any time.

We may also remove several of our assumptions to make our scalability argument more general. Not every server that replicates pages must be

willing to accept pages in return, as long as a sufficient number of push-cache servers exists. In the case of an interactive television network, for example, it might make sense to deploy many push-cache servers to cache replicated objects and only a few digital libraries which each store all the films available.

We may also remove the assumption that servers replicating pages are equally popular, although it would be possible in this case for a server to gain more load than it saved. In such a case the server would need to protect itself by refusing to accept new objects once its load has risen sufficiently.

Distributing the load created by popular items helps the Web scale as its population grows, but it creates a potential bottleneck at the primary server for two reasons. Clients must currently use the primary server in order to locate nearby replicas, and replicas must use the primary server to maintain consistency. We will discuss cache consistency in the next chapter; resource location places less of a burden on the primary server than serving files, since a single redirect messages can prevent all future references from a given client,

The last potential bottleneck to push-caching is the registry. The registry is essentially a database of every available push-cache server, and it must be able to handle requests arriving constantly from all push-cache server on the Internet. The registry must therefore be able to scale freely, up to millions of clients, and must be very reliable. This need for scalability and reliability implies that the registry must be a widely-replicated distributed database. Luckily, however, the registry does not need tight consistency. Because the registry selects lists of servers at random, a given instance of the database does not need to know about all the available servers in order to provide a representative sample, just most of them.

As long as updates propagate quickly enough among the instances of the registry such that they do not frequently include inactive servers in their lists of available servers, the system will function smoothly. The *refdbms* distributed bibliographic database system [17] serves as a good model for such a distributed database.

Chapter 6

Consistency Control

No discussion of information caching is complete without a discussion of appropriate cache consistency mechanisms. Caching information, whether by push-caching, proxy-caching, or client-caching, is useless unless there is some way to update the cached information in an efficient manner when the original data changes. These mechanisms insure that when original data change, cached copies of that data are eventually updated to once again mirror the original data.

There are several mechanisms currently in use on the Internet to maintain cache consistency: time-to-live fields, invalidation protocols, and client polling. Time-to-live fields (TTL) are easy to use, but are inaccurate. Each cached copy is associated with a TTL, such as 2 days or 12 hours. When the TTL expires the data is considered invalid and must be reloaded. It is hard to tell in advance for how long an object will be valid; therefore, it is necessary to set the TTL field to a relatively short interval and reload the object frequently to avoid returning stale data.

Client polling describes the scheme used with the Alex system, (see section 2.1.2. Client polling depends on the fact that the older a file is the less likely it is to change, and therefore the less often the client needs to check to see if its copy is stale. Specifically, the Alex scheme uses an *Update Threshold* to decide when an object should be considered stale. If `time since last checked` > `total age` × `update threshold` then the file is considered invalid. The example we provided in section 2.1.2 was for an update threshold of 10%. In such a case, if a file is 1 month old, then Alex will serve the file for up to three days (10% × 30 days = 3 days) before checking to see if it has become invalid.

Invalidation protocols are required when loose consistency is not sufficient; many distributed file systems rely on invalidation protocols to insure that cached copies do not become stale. Invalidation protocols depend on the server keeping track of cached copies of its data; when a file changes the server notifies caches that their copies are no longer valid.

One problem with invalidation protocols is that they have large non-trivial costs. Servers must keep track of where their objects are currently cached. This could be a prohibitive limit to a system's scalability unless caches are organized hierarchically, such that the server only has to keep track of the highest level caches. Invalidation protocols must also deal with unavailable clients as a special case: if a cache is temporarily unavailable then the server must continue trying to reach it; with no other mechanism for invalidation available the cache will not know to invalidate the object unless it hears from the server.

6.1 Related Work

Kurt Worrell, as part of his master's thesis [37], examined the use of TTL fields on the World Wide Web and determined through simulation that they are not appropriate for the Web because they return too much stale data. We found several inaccurate assumptions in his work, and later in this chapter we challenge his findings.

We mentioned above that caches must be organized hierarchically in order for an invalidation protocol to scale; Worrell found it necessary to make the further assumption that the *inclusion* principle holds for this hierarchy. The inclusion principle states that any object located in a lower level cache must also reside in all higher level caches in the hierarchy. This could lead to an inefficient use of disk space, since all objects cached on lower levels must also be cached on higher levels. Due to the nature of hierarchical caches, this means that one high level cache might potentially need to cache the equivalent of hundreds of low level caches. Otherwise pages may be kicked out of a low level cache simply because a high level cache is full, and it may be impossible to fully utilize space available in low level caches due to a lack of space in a high level cache.

Client-directed cache consistency mechanisms remove this requirement since there is no need to organize caches hierarchically. This also means that caches can be created and removed quite easily since there is no need to fit them into a hierarchical tree.

Worrell showed that invalidation protocols are superior to the use of TTL fields by simulating the performance of both a TTL-based cache consistency scheme and an Invalidation Protocol based consistency scheme. He showed that when the TTL field is set such that the two schemes are equivalent in their network bandwidth requirements, that 20% of the TTL requests are returned with stale data versus 0% for the invalidation scheme.

Worrell did not, however, simulate the Alex cache consistency scheme. Since we believed that the Alex scheme would perform more efficiently than the other schemes we wanted to correct this oversight, and Worrell was gracious enough to provide us with his simulator.

6.2 Cache Consistency Simulator

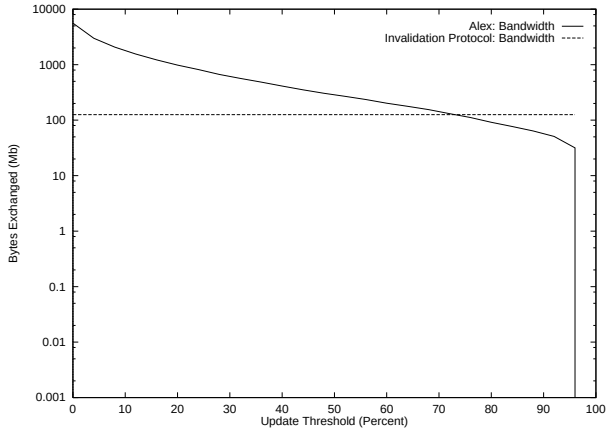
To evaluate the performance of the Alex cache consistency scheme we added it to the simulator used by Worrell. Prior to his evaluation of consistency protocols, he had gathered several months of data on the history of 4000 files located around the Web. This data recorded the average age of each file as well as its variance. He used this data to change the files over time, and he drove the simulator through a random stream of file references.

His simulator was also used to test the effects of a hierarchical cache on cache consistency performance; in order to isolate the effects of the cache consistency policy from the effects of stacking caches in a hierarchy, we remodeled the hierarchy of his caches in order to simulate a single cache. We therefore could not record $bytes \times hops$ as Worrell could; instead we simply recorded the number of bytes used by a cache in maintaining consistency, including invalidation messages, stale-data checks, and transferring files.

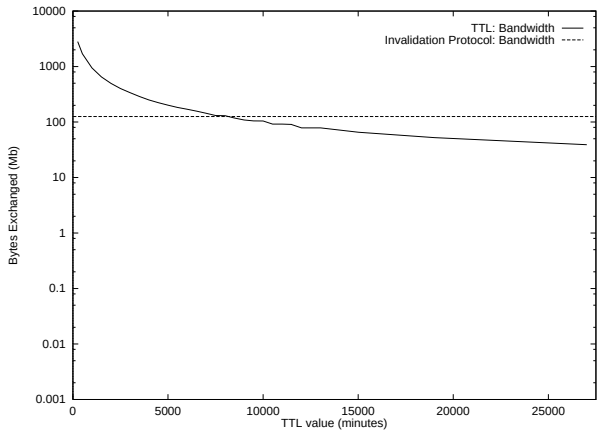
Worrell did not update objects with his invalidation protocol the instant they became invalidated, but instead simply marked them as invalid. Only when a client actually requested an invalidated object was it transferred from the primary server. This increases latency but decreases bandwidth.

6.2.1 Results

The results of simulating the Alex protocol against the TTL protocol are shown in Figure 6.1 and Figure 6.2. These graphs clearly show the tradeoffs involved in selecting a cache consistency protocol: invalidation consumes the optimal amount of bandwidth needed to insure no stale cache hits, but if one is willing to accept a certain number of stale hits it is possible to reduce the bandwidth requirements.

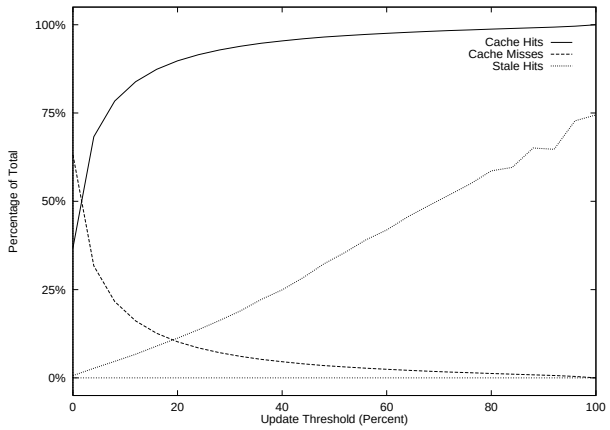


(a) Alex Cache Consistency Scheme

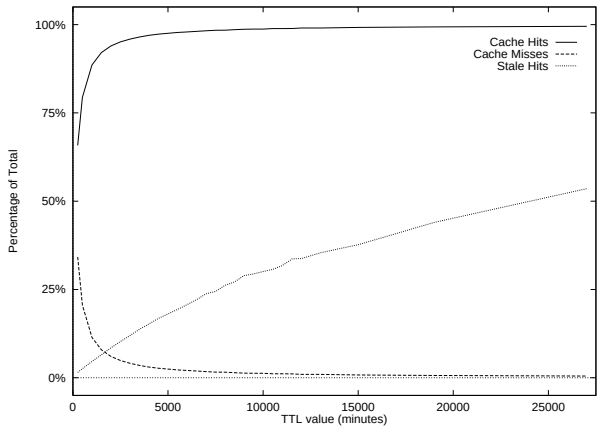


(b) Time-to-Live Fields

Figure 6.1: **Comparison of bandwidth used by invalidation protocol, Alex cache consistency scheme, and Time-To-Live fields.** The cache is pre-loaded with valid copies of all the files held in the primary server. Note the use of a log-scale to display the bandwidth with higher accuracy. Note also, however, the discontinuity on the far right of (a); this is an artifact of the y-axis log-scale, since a value of 0 on the y-axis is infinitely far away.



(a) Alex Cache Consistency Scheme



(b) Time-to-Live Fields

Figure 6.2: **Comparison of the hit rates of the Alex scheme to the hit rates of the TTL scheme.** Note that the stale hit rate rises as time-to-live fields become longer and as the Alex age threshold rises. The invalidation protocol always results in a 0% stale hit rate.

By comparing the two Figures one notes that for a given stale cache hit percentage, the TTL scheme uses nearly half the bandwidth of the Alex scheme. We had expected the Alex scheme to be more efficient than the TTL scheme; we address this discrepancy later in this chapter.

6.2.2 Conditional Retrieval

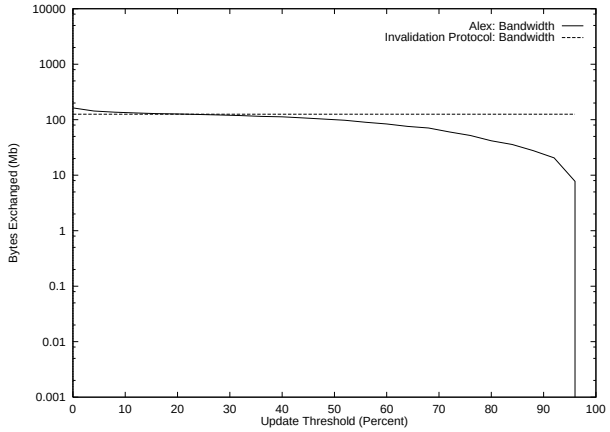
These results are comparable to those produced by Worrell, but we felt it was possible to improve the performance of TTL and Alex. We modified the simulator so that an invalid object would not be automatically retrieved when it was requested, but instead the cache would check first to make sure that the object is really invalid. If the object is still valid it is not removed from the cache, but instead is once again marked as valid. This improves performance because the Alex and TTL schemes were frequently invalidating and replacing objects that were not actually stale.

The effect of this change on network traffic is shown in Figure 6.3, and the effect of this change on cache hits and misses is shown in Figure 6.4. TTL still creates less network traffic than Alex, but the two are both more competitive relative to the invalidation protocol. Nevertheless, choosing a TTL of 5000 for example only saves 30% of the invalidation protocol's bandwidth but results in a 17% stale cache hit rate. This might not be acceptable for the moderate bandwidth savings.

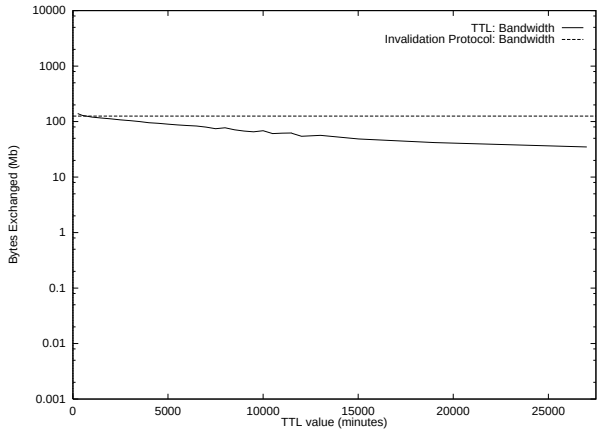
6.3 Trace-driven Simulator Modifications

We had expected that an adaptive protocol like the Alex protocol would do better than the static TTL protocol, so we examined the factors that contributed to its performance. In examining Worrell's simulator closely we realized that he was calculating file lifetimes with a flat distribution across the minimum and maximum he had observed. This scheme meant that there was no correlation among the series of generated lifetimes for a file. Our own observations of server traces taken from our modified Web server show that this is not the case; frequently a file will go unmodified for a long time, then it will be modified many times within a short time-period as the file is updated, and then it will be unmodified again for a long period of time.

An adaptive protocol like Alex will work well with a file exhibiting this behaviour; while the file is changing rapidly Alex will also check rapidly. Once the file stabilizes Alex will not check consistency as often.

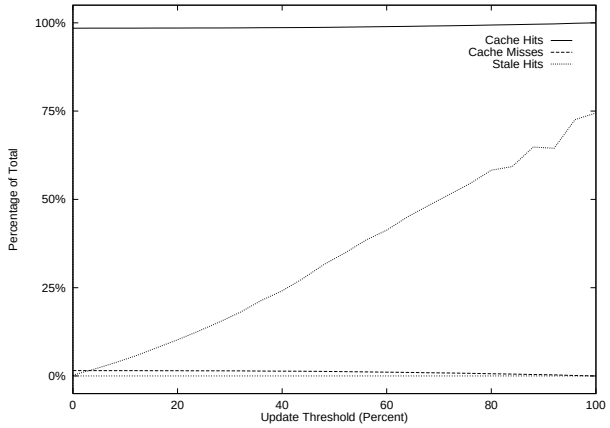


(a) Alex Cache Consistency Scheme

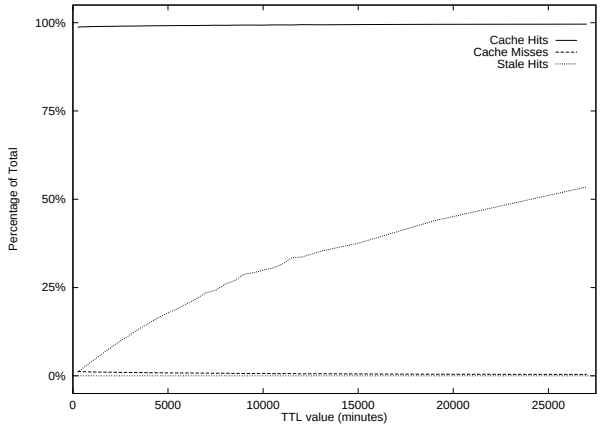


(b) Time-to-Live Fields

Figure 6.3: **Comparison of bandwidth used by invalidation protocol, Alex scheme, and TTL scheme.** Files are only transmitted when they are truly stale. Note that TTL and Alex now use less bandwidth than the Invalidation Protocol.



(a) Alex Cache Consistency Scheme



(b) Time-to-Live Fields

Figure 6.4: **Comparison of cache hit rates for Alex and TTL.** Notice the dramatic improvement on cache hits and misses, since files are not deleted from the cache unless they are truly stale. Notice also that the stale cache hit rate is unaffected by the change.

Server	Files	Requests	% Remote Requests	Total Changes	% Mutable Files	% Very Mutable Files
DAS	1403	30,093	84%	321	6.83%	2.71%
FAS	290	56,660	39%	11	2.41%	0.00%
HCS	573	32,546	50%	260	23.3%	5.22%

Table 6.1: **Summary of mutability statistics for various campus servers over a one-month period.** Mutable files changed more than once over the time period. Very mutable files changed more than 5 times. Any request that was not generated by a client in the `harvard.edu` domain was considered remote, and any files added in the middle of this time period were not included in these statistics. Notice that the most popular server, the FAS server, is also the one with the fewest mutable files.

Bestavros provides further reasons why this random lifetime generation is flawed; he found that on a given server only a few files change rapidly [2]. Furthermore he notes that globally popular files are the least likely to change, evidence that Worrell's random file selection is flawed as well. If file request distribution is skewed toward the most popular files then the overall number of stale hits would go down even further.

6.3.1 Trace Analysis

To test these results we modified Worrell's simulator to use actual Web server logs in generating file lifetimes. These were taken from our campus Web servers modified to store *last-modified* timestamps, described in section 3.1. A script reads through these access logs generating another trace that records every time a file changes.

Information pertaining to file changes from the three Web server traces is summarized in table 6.1. Information about the traces in this table confirms Bestavros' observation that the files with the greatest popularity are also the ones with the smallest mutability.

These files also change far less often than the files with randomly generated lifetimes. One run, for example, with randomly generated lifetimes involved 2085 files over a 56 day simulated run. Those 2085 files changed 19,898 times for an average probability that a given file would change per day of 17%. Our HCS trace on the other hand, the trace that changed the

most frequently, involved 573 files over 25 days. Those files changed a total of 260 times for an average probability that a given file would change per day of 1.8%. This result is confirmed by Bestavros, who found an average file-change probability per day between 0.5% and 2.0%, with more popular files changing less often than other files. Bestavros collapsed all changes made on a single day into one change; if we followed suit our probability drops even lower.

This represents an order of magnitude difference in rapidity of change; we expect therefore to obtain far fewer stale cache hits with the Alex and TTL protocols using the trace data than we did with Worrell's random lifetime generation.

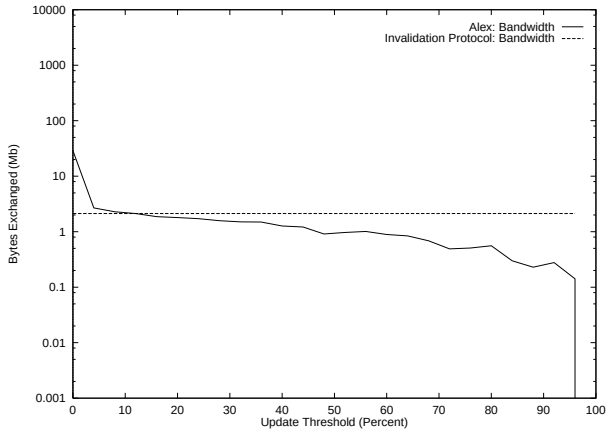
6.3.2 Results

We used these traces to drive the modified simulator, and we found that our hypothesis was confirmed. Since the files in our traces changed far less often than the files with randomly generated lifetimes, the stale hit count was therefore much less. We simulated the performance of an invalidation protocol, Alex, and TTL for each of our three traces, and then averaged the results together.

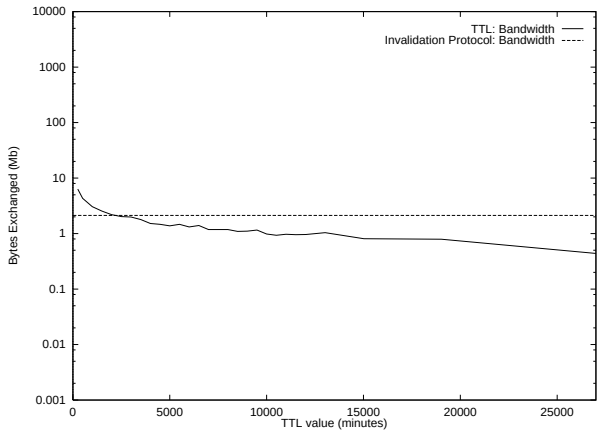
Figures 6.5 and 6.6 display the results. These graphs differ significantly from those generated by Worrell's random file-length generation, but surprisingly are still inconclusive. Neither Alex nor TTL yield many stale cache hits, but their bandwidth savings over an invalidation protocol is not significant either. Figure 6.7 shows the number of server operations generated by the various protocols: these include requests for documents, queries to determine whether documents are stale or not, and the invalidation of documents by the server. These graphs are also inconclusive: Alex is more efficient than TTL, but neither is superior to an invalidation protocol.

6.4 Conclusion

The results from this section are surprisingly ambivalent given Worrell's strong preference for an invalidation protocol. None of the three protocols performed efficiently in all cases, reflecting the tradeoffs discussed in chapter 6. Surprisingly, an adaptive protocol like Alex was not much more efficient than a static protocol like TTL. An invalidation protocol was very competitive in terms of bandwidth consumed and server load imposed, but

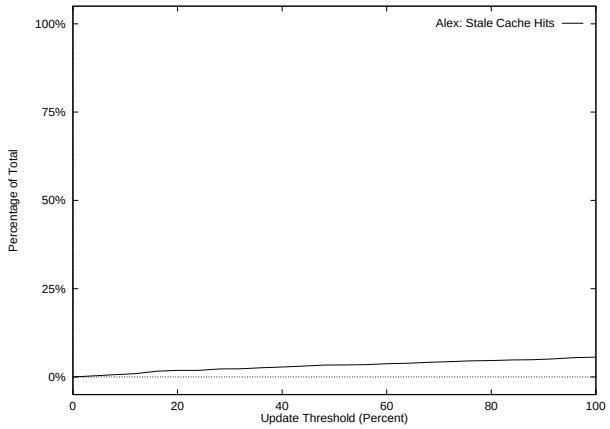


(a) Alex Cache Consistency Scheme

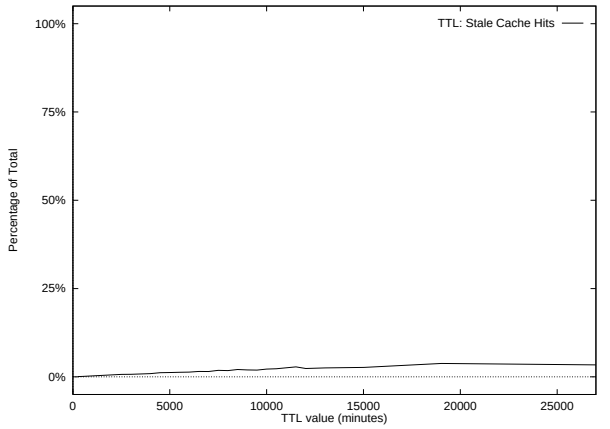


(b) Time-to-Live Fields

Figure 6.5: **Comparison of bandwidth used by invalidation protocol, Alex scheme, and TTL scheme from trace-driven simulations.** Shown is the average from the FAS trace, the HCS trace, and the DAS trace. Only those files that were already in the primary host at the beginning of the month were simulated. Note that the Alex scheme and the TTL scheme both use less bandwidth than the Invalidation Protocol.

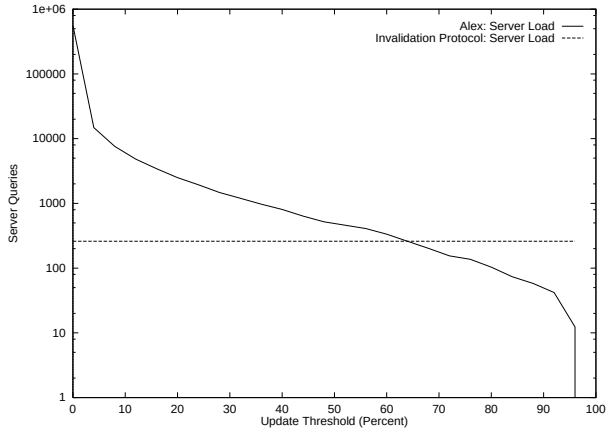


(a) Alex Cache Consistency Scheme

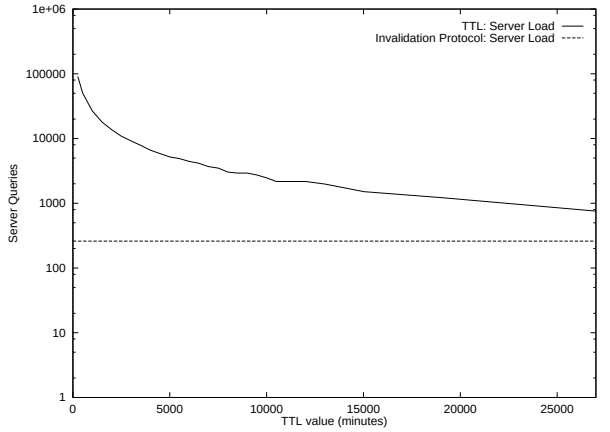


(b) Time-to-Live Fields

Figure 6.6: **Comparison of cache hit ratios from Alex scheme and TTL scheme for trace-driven simulations.** Note that either scheme results in a low stale data return rate when driven by real-world traces.



(a) Alex Cache Consistency Scheme



(b) Time-to-Live Fields

Figure 6.7: **Comparison of server loads imposed by the invalidation protocol, the Alex scheme, and the TTL scheme.** Notice that Alex imposes less load on the server than TTL, but that neither is superior to the invalidation protocol.

its other drawbacks, such as requiring the inclusion principle, make it difficult to implement on a wide-area system such that it can scale appropriately.

Under the real-world conditions such as those simulated by our server traces, the Alex and TTL scheme do not perform as poorly as Worrell claims. Biases in his simulation skewed his results in favor of an invalidation protocol; removal of the biases revealed that no protocol was overwhelmingly superior to the other. We conclude therefore with the counter-intuitive statement that a cache consistency protocol for use in a wide-area distributed system may be selected based on its ease of implementation and simplicity rather than on its bandwidth consumption and imposed server load.

We therefore recommend the Alex protocol for use in distributed Web caches because its performance on real world data is slightly better than TTL, and it is much easier to implement than an invalidation-based protocol because it is not necessary to worry about disconnected networks. If the client can not reach the server to update its invalidated objects it merely continues checking every time the object is requested. No additional state needs to be stored in either the client or the server to handle the error case.

Chapter 7

Conclusion

Our simulations showed that push-caching complements client-caching and proxy-caching in addressing the scalability problems plaguing the World Wide Web, but we must also acknowledge that push-caching is most effective when the full Internet topology is known. Geographical distance can help to predict topology, but the more accurate hop-based metric consistently out-performed the miles-based metric by at least 25%. Nevertheless, there are a number of potential applications (like proposed interactive television systems) where network topology is known and where push-caching's ability to quickly and dynamically distribute information in response to shifting popularity will be critical.

This is the first time that push-caching has even been investigated on a large-scale information system like the Web, and we are excited by its promise for solving not only Internet scalability problems but also for reducing network traffic and server load on any large-scale distributed system. As more and more computers become networked the need for efficient, scalable, and dynamic load-balancing and replication algorithms will increase. Push-caching is promising because it meets all of these criteria without requiring network architecture extensions or human intervention. We hope that the ease with which a server-initiated caching architecture may be deployed will lead to its rapid acceptance and integration into today's large-scale distributed systems.

We are also excited by the results of our cache consistency research, since by correcting the mistaken assumptions of earlier research we have shown that weak-consistency protocols are effective on the World Wide Web. This finding will reassure Web developers that the weak-consistency assumption

is correct, and by verifying that the simpler protocols are equally effective we have removed barriers to the introduction of widespread caching on the Web.

7.1 Future Work

There are still a large number of topics regarding server-initiated caching that we have not yet had time to explore in depth. These include server conflicts, and push-cache server management. We are especially interested in a more efficient mechanism for resource discovery so that clients do not have to query the original server to locate documents. There is no reason why an unavailable primary host should limit access to a file that is cached in a nearby push-cache server; this is one problem we look forward to solving.

Another area that we have not yet explored completely is the relationship between push-caching and client-caching. It has been argued that client-caches are most effective when caching lots of small, popular objects. Larger objects not only take up a disproportionate share of precious cache space, but latency for accessing large objects is often less important than for accessing smaller objects. We have already seen that push-caching is more efficient than client-caching at taking advantage of cache space; push-caching may be ideally suited to handling the distribution and replication of larger objects, leaving client caches to specialize in smaller ones. This is also an area that we are very excited about pursuing.

Finally, we plan to build a push-caching Web server, as well as a push-cache aware Web proxy. With these two tools we will explore the use of server-initiated caching in the real world, to measure its true effectiveness at reducing server load and network bandwidth. There are a number of servers and proxies with publicly available source code that would serve as suitable foundations for such work.

We also plan to build graphical tools that will allow system administrators to monitor the status of their push-caching web server visually. These tools will display the current location of object replicas against geographical maps, and will graphically illustrate push-cache dispersion.

7.2 Availability

The simulator, traces, and tools used for this project are available on the World Wide Web at `http://das-www.harvard.edu/research/vino/`. We

welcome any questions, comments, or criticisms.

7.3 Appreciation

This thesis would not have happened without the gentle nudging of my thesis advisor Margo Seltzer, whose advice, criticism, and praise was invaluable. She cheered me up when my spirts were lagging, and she was always there to answer any questions I might have about doing research. More than helping me simply write a paper, she quickly became a mentor helping indoctrinate me into the world of the research-scientist.

I would like to thank Michael Schwartz, Kurt Worrell, James Guyton, and Kim Claffy for making their data and code available to me. There is no way I would have been able to finish this thesis on time if I had not been able to leverage their work. I would like to thank Charlie Catlett and NCSA for providing me with access to the NCSA server logs, and I would like to thank Eugene Kim, HASCS, and the HCS for running my modified HTTPD server and for providing me with their server access logs.

Special gratitude goes to my roommates and to my fiancée Sarah who have had to live with push-caching and me for more than a year. I promise them that they will never again have to endure dinner conversations focused solely on autonomous replication and the future of the Internet (unless I decide to attend graduate school). Finally, thanks to Sarah, Diane, Jason, and Tom for proof-reading this thesis and for making editing suggestions.

Appendix A

Group Communication

Another area of distributed system research that has contributed toward the study of large-scale autonomous replication is that of group communication. A group communication system provides the ability for one member of a group to efficiently broadcast a message to all other members of the same group. This ability is most useful for distributed systems sharing information that changes rapidly and for systems where the data cannot reside in one centralized data base but must be distributed across multiple computers.

Several of these systems take network topology into account when making decisions on how to replicate information; this research has bearing on push-caching which also tries to make optimal caching decisions based on network topology. Group communication is also an efficient way to maintain shared information, such as that needed to maintain a listing of available push-cache servers.

One system, built by Richard Golding as his PhD thesis [16], is particularly efficient over traditional Internet point-to-point communication and can take network topology somewhat into account as it decides how to propagate messages. After discussing Golding's ideas, we will see how they have been adapted by Danzig et al for maintaining the shared databases used by replicated services.

A.1 Weak Consistency Group Communication

Golding's system provides a mechanism to allow a collection of principals to broadcast messages to each other using a weak consistency model such that divergent state eventually converges. His framework includes message

delivery, message ordering, group membership routines, and support for applications that need these services.

The protocol he introduces to accomplish these tasks he calls Timestamped Anti-Entropy (TSAE). TSAE uses delayed communication between principals to efficiently batch messages in a queue and deliver them later. Pairs of principals periodically take part in an *anti-entropy* session where they contact each other to exchange messages. This system scales well because these anti-entropy sessions can take place in parallel.

A principal must be able to survive temporary failures and host crashes, and must be equipped with stable storage to record information between network failures.

To efficiently exchange information with other members of the group, principals must be able to locate group members near them. This requires some sort of name or location service, as well as a performance prediction service [15] to order principals by locality. Performance prediction uses latency, failure rate, and bandwidth to determine expected performance. It does not use the number of network hops or expected backbone savings, and therefore does not require a prior knowledge of network topology. It can be argued that fast networks make expected backbone savings a more important metric than user latency, but it could also be argued that latency is the only viable metric that can be easily determined.

Performance prediction is built on top of the *ping* program, using ICMP echo to test the latency between hosts. His results are encouraging: for most hosts previous latency is a good predictor of future performance. Only for hosts very far away or some hosts overseas is this not necessarily the case. For the most part, however, the latency variance is small, and as we show in section 3.3, latency is proportional to the number of network hops. We focus primarily on geography in our own work because determining latency between two arbitrary hosts requires running ping from one of them, and such a step would prohibit one server from calculating the optimal solution without enormous overhead.

The TSAE protocol guarantees that any knowledge owned by one principal will eventually spread to all other principals, using the anti-entropy session. This session is quite simple: one principal picks another principal and establishes a session between them. The two principals then exchange any messages that the other does not know about, and disconnect. This protocol is guaranteed to converge eventually so that inconsistency is always finitely bounded.

The efficiency of the protocol is determined by the session partner selec-

tion process. Golding presents simulation results for a variety of different selection policies, such as random selection or distance- based selection, and considers their impact on network traffic and propagation time. The time to propagate a message using a uniform random partner selection policy scales well with the size of the group. Five sites takes about four mean entropy-time intervals, while 160 sites only takes twice that. The best time is achieved, however, with age-based partner selection. Fixed topologies in general are poor, although adaptive topologies work well.

Network traffic is another important factor. Each message is sent exactly once to each principal; therefore the total amount of network traffic generated is determined by the network topology and the partner selection policy. He introduced new partner selection policies to try and reduce the amount of total network traffic such as cost-biased and cost-squared biased.

Golding's TSAE protocol is already finding use in the Internet; besides the *refdbms* system [17] that he helped design, Danzig et al have explored using the TSAE to help handle massively replicated services [11]. This research is particularly important because a push-cache server will need a system to track available servers, and it may be necessary to distribute this service to insure that it is scalable.

A.2 Service Replication

Danzig et all designed an architecture that supports a large group of autonomous servers and provides a way for them to maintain a shared distributed database efficiently. Their architecture uses a *flood-d* protocol that is based on Golding's TSAE protocol to propagate database updates. Internet Multicast is used to broadcast packets on the Internet.

The key contribution of this paper is the *flood-d* protocol. Designed to scale to thousands of autonomous hosts, it groups service replicas into multiple, autonomously administered replication groups. This has the effect of reducing the amount of state each replica must keep. Service replicas continuously measure Internet bandwidth and use this information to create optimal update topologies.

The approach used to create these optimal update topologies is attractive because it actually solves for an optimal update topology. They use Steiglitz's algorithm to determine a logical connectivity topology with node connectivity k, a minimal diameter, and a minimal edge cost. As long as $k > 1$ there is some protection against network failures. They approxi-

mate an optimal solution efficiently using a stochastic algorithm known as simulated annealing.

Their simulations show that this approach is efficient. Gathering replicas into groups results in faster update propagation, and taking advantage of the Internet topology in picking neighbors reduces the cost for propagating updates. We expect that using topology information in selecting cache locations will also result in bandwidth and latency savings as well.

Bibliography

[1] T. Berners-Lee, R. Cailliau, J-F. Groff, and B. Pollermann. World-wide web: The information universe. *Electronic Networking Research, Applications and Policy*, 2(1):52–58, 1992.

[2] Azer Bestavros. Demand-based document dissemination for the world-wide web. Technical Report 95-003, Boston University, 1995.

[3] Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, April 1982.

[4] Matthew A. Blaze. Caching in large-scale distributed file systems. Technical Report TR-397-92, Princeton University, January 1993.

[5] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Mich ael F. Schwartz. Harvest: A scalable, customizable discovery and access system. Technical Report CU-CS-732-94, University of Colorado, Boulder, 1994.

[6] C. Mic Bowman, Peter B. Danzig, Udi Manber, and Michael E. Schwartz. Scalable Internet resource discovery: Research problems and approaches. *Communications of the ACM*, 37(8):98–107, August 1994.

[7] Hans-Werner Braun and Kimberly Claffy. Web traffic characterization: an assessment of the impact of caching documents from ncsa's web server.
In *Electronic Proceedings of the Second World Wide Web Conference '94: Mosaic and the Web*, Chicago, IL, October 1994. Available from http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/DDay/claffy/main.html.

[8] Vincent Cate. Alex - A global filesystem. In *USENIX File Systems Workshop Proceedings*, pages 1–12, Ann Arbor, MI, May 21 - 22 1992. USENIX.

[9] Ellis S. Cohen. Boston restaurant list. http://www.osf.org:8001/boston-food/boston-food.html.

[10] Peter Danzig, Richard Hall, and Michael Schwartz. A case for caching file objects inside internetworks. Technical Report CU-CS-642-93, University of Colorado, Boulder, 1993. available from 'ftp.cs.colorado.edu' in 'pub/cs/ techreports/ schwartz/'.

[11] Peter Danzig, Dante DeLucia, and Katia Obraczka. Massively replicating services in wide area internetworks. Technical Report 93-541, University of California at Santa Cruz, 1993.

[12] Peter Danzig, Katia Obraczka, and Anant Kumar. An analysis of wide-area name server traffic. Technical Report 92-504, University of California at Santa Cruz, 1992.

[13] Glenn Davis. Cool site of the day. http://www.infi.net/cool.html.

[14] Alan Emtage and Peter Deutsch. Archie - an electronic directory service for the internet. In *Proceedings of the USENIX Winter Conference*. USENIX, January 1992.

[15] Richard A. Golding. End-to-end performance prediction for the internet (work in progress). Technical Report UCSC-CRL-92-26, University of California, Santa Cruz, 1992.

[16] Richard A. Golding. *Weak-consistency group communciation and membership*. PhD thesis, University of California, Santa Cruz, 1992.

[17] Richard A. Golding, Darrell D. E. Long, and John Wilkes. The /em refdbms distributed bibliographic database system. In *1994 Winter USENIX*, pages 47–62, San Francisco, CA, January 17-21 1994. USENIX.

[18] James D. Guyton and Michael F. Schwartz. Experiences with a survey tool for discovering network time protocol servers. In *1994 Summer USENIX*, pages 257–265. USENIX, June 1994.

[19] James D. Guyton and Michael F. Schwartz. Locating nearby copies of replicated internet servers. Technical Report CU-CS-762-95, University of Colorado at Boulder, 1995.

[20] White House. Welcome to the white house. http://www.whitehouse.gov/.

[21] V. Jacobsen. Traceroute software. Lawrence Berkeley Laboratories, December 1988. Available at ftp://ftp.ee.lbl.gov/pub/traceroute.tar.Z.

[22] Saumel J. Leffler, Marchall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley Publishing Company, 1989.

[23] Daniel E. Lenoski. tthe design and analysis of DASH: a scalable directory-based multiprocessor. Technical Report CSL-TR-92-507, Stanford University, January 1992.

[24] Tom Libert. telnet://martini.eecs.umich.edu:3000/.

[25] Ari Luotonen and Kevin Altis. World-wide web proxies. In *Computer Networks and ISDN systems*. First International Conference on the World-Wide Web, Elsevier Science BV, 1994. available from 'http://www.cern.ch/ PapersWWW94/ luotonen.ps'.

[26] B. Lyon, G. Sager, J. M. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, D. Walsh, and P. Weiss. Overview of the sun network file system. Technical Report CMU-CS-84-137, Sun Microsystems, Inc., January 1985.

[27] Inc. Sun Microsystems. NFS: Network file system protocol specification. RFC - 1094, Network Information Center, SRI International, March 1989.

[28] J.H. Morris, M. Satyanarayanan nad M.H. Conner, J.H. Howard, D.S.H. Rosenthal, and F.D. Smith. Andrew: a distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, March 1986.

[29] Mosaic-x@ncsa.uiuc.edu. Using proxy gateways. World-Wide Web. available from 'http://www.ncsa.uiuc.edu/ SDG/Software/ Mosaic/ Docs/ proxy-gateways.html'.

[30] nic.merit.edu. ftp://nic.merit.edu/nsfnet/ announced.networks/nets.unl.now.

[31] Nicolas Pioch. Le weblouvre. World-Wide Web. http: //mistral.enst.fr/ pioch/louvre/louvre.shtml.

[32] National Weather Service Ski Report. New england alpine ski report. http://www.ksu.edu/unicorn/albrec.weather.

[33] Mirjana Spasojevic, Mic Bowman, and Alfred Spector. Information sharing over a wide-area file system. In *Electronic Proceedings of the Second World Wide Web Conference '94: Mosaic and the Web*, Chicago, IL, October 1994.

[34] Charles L. Viles and James C. French. Availability and latency of world wide web information servers. *Computing Systems*, 8(1):61–91, 1995.

[35] Randolph U. Wang and Thomas E. Anderson. xfs: A wide area mass storage file system. Technical Report CSD-93-783, University of California, Berkeley, http://cs-tr.cs.berkeley.edu/ TR/ UCB:CSD-93-783, 1993.

[36] David C. M. Wood, Sean S. Coleman, and Michael F. Schwartz. Fremont: A system for discovering network characteristics and problems. In *1993 Winter USENIX Proceedings*, pages 335–348, Ann Arbor, MI, January 1993. USENIX. available from 'ftp.cs.colorado.edu' as: 'pub/cs/techreports/schwartz/PostScript/Fremont.ps.Z'.

[37] Kurt Jeffery Worrell. *Invalidation in Large Scale Network Object Caches*. PhD thesis, University of Colorado–Boulder, 1994.

Glossary

Archie A popular service on the Internet that provides a directory of files available over the Internet. Archie is so popular that a simple search frequently takes several minutes; there is no easy way to make Archie more accessible short of manually replicating it onto several computers, and this requires significant effort. Push-caching is one approach that would allow Archie to automatically scale to meet the demands of the Internet.

Backbone The Internet backbone connects the various regional networks with an extremely high-speed connection. There are now several backbones on the Internet; these backbones are connected at only a few points which easily become congested. Most data sent across the Internet must eventually flow over a backbone, so most wide-area caching schemes focus on reducing backbone traffic. See Internet for more information.

Bandwidth Bandwidth describes the amount of information that flows between two computers across a network. Networks are limited in their available bandwidth; if computers try to send too much data across a network a traffic jam will occur and no data will get through. This fact motivates current research on reducing the bandwidth needs of various Internet services like the World Wide Web.

Browser Software that lets a user navigate across the World Wide Web, connecting to Web servers and displaying their information.

Byte Basic unit of computer storage. A byte can store a single character.

Cache, Caching Caching describes the act of copying data in order to make it easier to access in the future. The space taken up by these

copies is called a cache; cache space is frequently limited. With client-caching, the client decides what data to copy and where to store them. With push-caching, the server decides what data to copy and where to store them. A cache miss occurs when a client requests a file that is not in the cache, and a stale cache hit occurs when a client is provided with a file from the cache that is out-of-date.

A good example of caching would be photocopying several pages out of a reference book. It is much more efficient to visit the libary once and make a personal copy of important information than it is to return to the library every time that information is needed. The example just described is an example of client-caching. An example of push-caching would be for the library to place copies of popular books in several convenient locations so that library patrons would not have to fight over them.

Client In many information transactions, one participant is usually designated the client, and the other participant the server. The client usually initiates the transaction, seeking information; the server is usually the repository for information that the client needs. A good example would be the Oracle at Delphi acting as the server; clients ask the Oracle questions and receive answers in return.

Firewall Firewalls are recent inventions; they limit the degree to which computers on one side of the firewall may interact with computers on the other side of the firewall. They protect corporations from hackers, but they also make it more difficult for corporate computers to access legitimate Internet services like the World Wide Web.

FTP File Transfer Protocol. This protocol allows two computers on the Internet to efficiently exchange a file, and until recently FTP traffic accounted for as much as 50% of the total Internet traffic.

Gb Gigabyte. Equivalent to 1024 Megabytes, or 1,073,741,824 bytes. A gigabyte can store approximately 670,000 pages of text, or roughly 256 copies of the King James Bible.

Hops, network See Internet.

Internet The Internet wires millions of computers together so that they may exchange information with one another. The Internet may be envisioned as an extremely fast postal service. Each computer on the

Internet has a unique address, like a mailing address. This unique address consists of two parts: the network on which the computer resides, analagous to a zip code, and the location of the computer on that network, analagous to a post office box. Information travels the Internet in the form of packets, similar to post cards since they are limited in the amount of information they can contain. In order to send a large file across the Internet it is automatically broken up into many small packets which are sent separately and then reassembled on the other end.

A computer connects to the Internet through a wire, known as a network connection, that is attached to another computer already on the Internet. This approach is relatively simple, but it means that when one computer sends a packet to another computer, that packet may have to travel through many other computers in between. There are often several different routes between two computers on the Internet; packets are routed relatively efficiently, but several packets may actually travel different routes between the same two computers. One measure of how far apart two computers on the Internet are is to measure how many other computers a packet must travel through to reach the final destingation, or how many *network hops* the packet must make.

Most computers on the Internet may be divided into three categories: they may be part of a university or corporate network, they may be part of a regional network, or they may be part of a national backbone. A packet travelling between a computer at Harvard to a computer at Stanford, for example, will first travel across the Harvard network until it reaches the New England regional network. It will travel across this network until it reaches a national backbone. It will travel across this backbone to California, and it will then travel across the Bay Area regional network until it arrives at the Stanford University network. Finally, it will travel across the Stanford network until it reaches its final destination.

Kb Kilobyte, equivalent to 1024 bytes. A kilobyte can store just over a half page of text.

Latency The amount of time that elapses between a request and its response.

Load Short for workload.

Mb Megabyte, equivalent to 1024 Kilobytes, or 1048576 bytes. A megabyte can store approximately 650 pages of text.

Metric A metric describes a value which is being measured. There are two metrics which may be used to measure how much information flows between two computers, for example: the number of bytes, or the number of bytes times the number of network hops.

NFS Network File System. A popular protocol created by Sun Microsystems that allows multiple workstations to share the same set of files.

Portable Software is portable if it can be easily rewritten to run on different computer systems.

Primary Host The primary host for a document on the World Wide Web is that document's owner. There may be many copies of a given document in circulation, but the primary host is the only computer that is allowed to make changes to the document, and therefore the primary host has the final say on whether a copy is up-to-date or is out-of-date.

Proxy Cache A proxy cache allows several clients to share the same cache space (see caching). To use a proxy cache a client requests all of its files from the proxy, not from the file's primary host. If the proxy cache has a copy of the desired file in its cache, then it returns the copy. Otherwise the proxy requests the file from the primary host, keeps a copy in its cache, and returns the document to the client.

Push-caching see caching.

Regional Network Regional networks form the glue between a campus-sized network (found at a university or corporation), and the backbone networks that transport data across the country and around the world. These regional networks are typically defined by a geographical area; NEARNet, for example, services New England, and BARNet services the Bay Area in California. See Internet for more information.

Replica A copy.

Scalability A system scales when it can grow to cope with increased demand.

Server A server typically provides information; clients connect to the server to use its resources. See Client for more information.

Topology, Network Describes how computers on a network are connected to one another. The number of network hops between two computers on the Internet can be derived from the appropriate network topology. See Internet for more information.

Traffic See bandwidth.

TCP Transmission Control Protocol. The Internet is inherently unreliable, and packets (see Internet) sent between two computers are frequently lost. To make up for this, most services on the Internet use the TCP protocol to insure reliable packet transmission. The TCP protocol will automatically retransmit lost packets, but setting up and taking down TCP connections takes longer than setting up a raw connection.

TTL Time-To-Live. A TTL defines for how long a given piece of data is expected to be valid; when the TTL expires the data is assumed to be invalid.

World Wide Web The World Wide Web defines a service offered over the Internet that allows computers to easily share complex information including text, pictures, and even video or music. Some computers are defined as Web servers; these machines make information available. Other computers are defined as Web clients; these machines connect to the servers that offer information and display this information for their users. Web servers can provide links to other Web servers; in this manner a client can effortlessly jump from one server to another without any concern for the amount of network traffic that is generated.

The Web is popular because of the vast amount of information available on the Web. Everything from classic literature to famous paintings to Boston movie listings and restaurant reviews are available on the Web, and currently the vast majority of this information is available for free. The Web is so popular that many servers can not keep up with the thousands of clients trying to access their information, and some clients therefore see very large delays in retrieving information. This thesis addresses this problem by allowing servers to distribute their workload across other servers.

xFS Experimental File System.